Powersoft®

Open Tools from Sybase, Inc.

PowerBuilder

PowerScript Reference: Volume 2

Version 6

PowerBuilder®

AA0836

October 1997

Contents

VOLUME 2

About This Book

Subject

This book describes syntax and usage information for the PowerScript language including variables, expressions, statements, events, and functions.

Audience

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

CHAPTER 10

PowerScript Functions

About this chapter

This chapter provides syntax, descriptions, and examples for PowerScript functions.

Contents

The functions are listed alphabetically.

Abs

Description

Calculates the absolute value of a number.

Syntax

Abs (*n*)

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

Return value

The data type of *n*. Returns the absolute value of *n*. If *n* is NULL, Abs returns NULL.

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 the *DataWindow Reference*

AcceptText

Description

Applies the contents of the DataWindow's edit control to the current item in the buffer of a DataWindow control or DataStore. 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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**AcceptText ( )**

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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). If *dwcontrol* is NULL, AcceptText returns NULL.

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 in the ItemChanged event
Calling AcceptText in the ItemChanged event has no effect.

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 container, allowing the user to work with the object using the server's commands.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE controls and OLE DWObjects (objects within a DataWindow object that is within a DataWindow control)

Syntax

objectref.**Activate** (*activationtype*)

Argument	Description
objectref	The name of the OLE control or the fully qualified name of a OLE DWObject within a DataWindow control that contains the object you want to activate The fully qualified name for a DWObject has this syntax: *dwcontrol*.Object.*dwobjectname*
activationtype (optional)	A value of the enumerated data type omActivateType specifying where the user will work with the OLE object. Values are: ♦ InPlace! — (Default) 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 For the OLE control, *activationtype* is required

Return value

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

-1 Container 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

If any argument's value is NULL, Activate returns NULL.

Examples

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

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

This example activates the OLE DWObject ole_graph in the DataWindow control dw_1 in the Microsoft Graph server application:

```
integer result
result = dw_1.Object.ole_graph.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*)

Argument	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. 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. If any argument's value is NULL, AddCategory returns NULL.

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. 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

AddColumn

Description

Adds a column with a specified label, alignment, and width.

Applies to

ListView controls

Syntax

listviewname.**AddColumn** (*label, alignment, width*)

Argument	Description
listviewname	The name of the ListView control to which you want to add a column
label	A string whose value is the name of the column you are adding
alignment	A value of the enumerated data type Alignment specifying the alignment of the column you are adding. Values are: ♦ Center! ♦ Justify! ♦ Left! ♦ Right!
width	An integer whose value is the width of the column you are adding

Return value

Integer. Returns the column index if it succeeds and -1 if an error occurs.

Usage

The AddColumn function adds a column at the end of the existing columns unlike the InsertColumn function which inserts a column at a specified location.

You must populate a ListView control with columns before you can display the contents in report view.

Examples

This script for a ListView event creates three columns in a ListView control:

```
integer index

FOR index = 3 to 25
   This.AddItem("Category " + String(index), 1)
NEXT

This.AddColumn("Name" , Left! , 1000)
This.AddColumn("Size" , Left! , 400)
This.AddColumn("Date" , Left! , 300)
```

See also

InsertColumn
DeleteColumn

AddData

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

To add data to	Use
Any graph type except scatter	Syntax 1
Scatter graphs	Syntax 2

Syntax 1

For all graph types except scatter

Description

Adds a data point to a series in a graph. Use Syntax 1 for any graph type except 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.**AddData** (*seriesnumber*, *datavalue* {, *categoryvalue* })

Argument	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

Long. Returns the position of the data value in the series if it succeeds and -1 if an error occurs. If any argument's value is NULL, AddData returns NULL.

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. 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 INFO 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)
```

See also

DeleteData
FindSeries
GetData
InsertData

Syntax 2

For scatter graphs

Description

Adds a data point to a series in a scatter graph.

Syntax

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

Argument	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

Long. Returns the position of the data value in the series if it succeeds and -1 if an error occurs. If any argument's value is NULL, AddData returns NULL.

Examples

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

AddItem

Adds an item to a list control.

To add an item to	Use
A ListBox or DropDownListBox control	Syntax 1
A PictureListBox or DropDownPictureListBox control	Syntax 2
A ListView control when you only need to specify the item name and picture index	Syntax 3
A ListView control when you need to specify all the properties for the item	Syntax 4

Syntax 1

For ListBox and DropDownListBox controls

Description

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

Applies to

ListBox and DropDownListBox controls

Syntax

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

Argument	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. If any argument's value is NULL, AddItem returns NULL.

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 property 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.

AddItem and InsertItem do not update the Items property array. You can use FindItem to find items added during execution.

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 property to TRUE. Otherwise, it may take longer than expected to add the items.

VBX controls If you have created a VBX user object for a VBX control that supports the AddItem method, use the AddItem or InsertItem function instead of its AddItem method.

Examples

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 property is FALSE, the statement above returns 3 (because Edit File becomes the third and last item). If the Sorted property 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

Syntax 2

For PictureListBox and DropDownPictureListBox controls

Description

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

Applies to

PictureListBox and DropDownPictureListBox controls

Syntax

listboxname.**AddItem** (*item* {, *pictureindex* })

Argument	Description
listboxname	The name of the PictureListBox or DropDownPictureListBox in which you want to add an item
item	A string whose value is the text of the item you want to add
pictureindex (optional)	An integer specifying the index of the picture you want to associate with the newly added item

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. If any argument's value is NULL, AddItem returns NULL.

Usage

If you don't specify a picture index, the newly added item will not have a picture.

If you specify a picture index that doesn't exist, that number is still stored with the picture. If you add pictures to the picture array so that the index becomes valid, the item will then show the corresponding picture.

FOR INFO For additional notes about items in list boxes, see Syntax 1.

Examples

This example adds the item Cardinal to the PictureListBox plb_birds:

```
integer li_pic, li_position
string ls_name, ls_pic

li_pic = plb_birds.AddPicture("c:\pics\cardinal.bmp")
ls_name = "Cardinal"
li_position = plb_birds.AddItem(ls_name, li_pic)
```

If plb_birds contains Robin and Swallow and the Sorted property is FALSE, the AddItem function above returns 3 because Cardinal becomes the third and last item. If the Sorted property is TRUE, AddItem returns 1 because Cardinal is first when the list is sorted alphabetically.

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
li_pic = plb_birds.AddPicture("HD:pics:cardinal.bmp")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
li_pic = plb_birds.AddPicture &
("/export/home/pics/cardinal.bmp")
```

See also

DeleteItem
FindItem
InsertItem
Reset
TotalItems

Syntax 3 For ListView controls

Description

Adds an item to a ListView control.

Applies to

ListView controls

Syntax

listviewname.**AddItem** (*label, pictureindex*)

Argument	Description
listviewname	The name of the ListView control to which you are adding a picture or item
label	The name of the item you are adding
pictureindex	The index of the picture you want to associate with the newly added item

Return value

Integer. Returns the index of the item if it succeeds and -1 if an error occurs.

Usage

Use this syntax if you only need to specify the label and picture index of the item you are adding to the ListView. If you need to specify more than the label and picture index, use Syntax 4.

Platform information
On Windows 3.1 (16-bit) there is a limitation on the total number of rows within a ListView control. The rule is as follows:

(no of rows) x [(image height in pixels) + (margin height in pixels)]<32,767

Examples

This example uses AddItem in the Constructor event to add three items to a ListView control:

```
lv_1.AddItem("Sanyo" , 1)
lv_1.AddItem("Onkyo" , 1)
lv_1.AddItem("Aiwa" , 1)
```

See also

DeleteItem

FindItem
InsertItem
Reset
TotalItems

Syntax 4

For ListView controls

Description

Adds an item to a ListView control by referencing all the attributes in the ListView item.

Applies to

ListView controls

Syntax

listviewname.**AddItem** (*item*)

Argument	Description
listviewname	The name of the List View control to which you are adding a picture or item
item	The item you are adding

Return value

Integer. Returns the index of the item if it succeeds and -1 if an error occurs.

Usage

Use this syntax if you need to specify all the properties for the item you want to add. If you only need to specify the label and picture index, use Syntax 3.

Platform information
On Windows 3.1 (16-bit) there is a limitation on the total number of rows within a ListView control. The rule is as follows:

(no of rows) x [(image height in pixels) + (margin height in pixels)]<32,767

Examples

This example uses AddItem in a CommandButton Clicked event to add a ListView item for each click:

```
count = count + 1

listviewitem l_lvi

l_lvi.PictureIndex = 2
l_lvi.Label = "Item "+ string(count)
lv_1.AddItem(l_lvi)
```

See also

DeleteItem
FindItem
InsertItem
Reset
TotalItems

AddLargePicture

Description

Adds a bitmap, icon, or cursor to the large image list.

Applies to

ListView controls

Syntax

listviewname.**LargePicture** (*picturename*)

Argument	Description
listviewname	The name of the ListView control to which you are adding a bitmap, icon, or cursor
picturename	The name of the bitmap, icon, or cursor you are adding to the large image list

Return value

Integer. Returns the picture index if it succeeds and -1 if an error occurs.

Usage

When you add a large picture to a ListView, it is given the next available picture index in the ListView. For example, if your ListView has two pictures, the next picture you add will be assigned picture index number 3.

Before you add large pictures, you can specify scaling for the pictures by setting the LargePictureWidth and LargePictureHeight properties. The dimensions in effect when you add the first picture determine the scaling for all pictures. Changing the property values after you add pictures has no effect.

If you do not specify values for LargePictureWidth and LargePictureHeight before you add pictures, the dimensions of the first image determine the scaling for all pictures you add.

When you add a bitmap, specify the color in the bitmap that will be transparent by setting the LargePictureMaskColor property before calling AddLargePicture. You can change the LargePictureMaskColor property between calls.

Examples

This example adds the file "folder.ico" to the large picture index of the ListView lv_files:

```
//Add large picture
integer index
index = lv_files.AddLargePicture("folder.ico")
```

See also

DeleteLargePicture

AddPicture

Description

Adds a bitmap, icon, or cursor to the main image list.

Applies to

PictureListBox, DropDownPictureListBox, and TreeView controls

Syntax

controlname.**AddPicture** (*picturename*)

Argument	Description
controlname	The name of the control to which you want to add an icon, cursor, or bitmap to the main image list
picturename	The name of the icon, cursor, or bitmap you want to add to the main image list

Return value

Integer. Returns the picture index number if it succeeds and -1 if an error occurs.

Usage

The picture is assigned an index in the order in which it is added to the control.

Adding pictures during execution does not update the PictureName property array. Because the picture is added at the end of the list, the return value from AddPicture is the number of pictures associated with the control.

Before you add pictures, you can specify scaling for the pictures by setting the PictureWidth and PictureHeight properties. The dimensions in effect when you add the first picture determine the scaling for all pictures. Changing the property values after you add pictures has no effect.

If you do not specify values for PictureWidth and PictureHeight before you add pictures, the dimensions of the first image determine the scaling for all pictures you add.

When a you add a bitmap, specify the color in the bitmap that will be transparent by setting the PictureMaskColor property before calling AddPicture. You can change the PictureMaskColor property between calls.

Examples

This example adds a picture to a TreeView control and associates it with a new TreeView item:

```
long ll_tvi
integer li_picture
li_picture = &
tv_list.AddPicture("c:\apps_pb\staff.ico")
ll_tvi = tv_list.FindItem(RootTreeItem!, 0)
tv_list.InsertItemFirst(ll_tvi, "Dept.", li_picture)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
li_pic = plb_birds.AddPicture("HD:PB Apps:staff.ico")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
li_pic = plb_birds.AddPicture &
("/export/home/apps_pb/staff.ico")
```

See also

DeletePicture

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 in which 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*)

Argument	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. If any argument's value is NULL, AddSeries returns NULL.

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

AddSmallPicture

Description

Adds a bitmap, icon, or cursor to the small image list.

Applies to

ListView controls

Syntax

listviewname.**AddSmallPicture** (*picturename*)

Argument	Description
listviewname	The name of the ListView control to which you are adding a small image
picturename	The name of the bitmap, icon, or cursor you are adding to the ListView control small image list

Return value

Integer. Returns the picture index if it succeeds and -1 if an error occurs.

Usage

When you add a small picture to a ListView control, it is given the next available picture index in the ListView. For example, if your ListView has two pictures, the next picture you add will have index number 3.

Before you add small pictures, you can specify scaling for the pictures by setting the SmallPictureWidth and SmallPictureHeight properties. The dimensions in effect when you add the first picture determine the scaling for all pictures. Changing the property values after you add pictures has no effect.

If you do not specify values for SmallPictureWidth and SmallPictureHeight before you add pictures, the dimensions of the first image determine the scaling for all pictures you add.

Before you call AddSmallPicture, specify the color in the bitmap that will be transparent by setting the SmallPictureMaskColor property. You can change the SmallPictureMaskColor property between calls.

Examples

This example adds the file "shortcut.ico" to the small picture index of the ListView lv_files:

```
//Add small picture
integer index
index = lv_files.AddSmallPicture("shortcut.ico")
```

See also

DeleteSmallPicture

AddStatePicture

Description

Adds a bitmap, icon, or cursor to the state image list.

Applies to

ListView and TreeView controls

Syntax

controlname.**AddStatePicture** (*picturename*)

Argument	Description
controlname	The name of the ListView or TreeView control to which you are adding a bitmap, cursor, or icon
picturename	The name of the bitmap, icon, or cursor you are adding

Return value

Integer. Returns the picture index if it succeeds and -1 if an error occurs.

Usage

For ListViews in large icon view, the state picture is a picture displayed to the left of the large picture, by default in a smaller size. For TreeViews, the state picture is displayed to the left of the regular picture and the item is moved to the right to make room for it.

If you specify either StatePictureWidth or StatePictureHeight, the picture is scaled to the size specified by that property.

When a you add a bitmap, specify the color in the bitmap that will be transparent by setting the StatePictureMaskColor property before calling AddPicture. You can change the StatePictureMaskColor property between calls.

Examples

This example adds the file "star.ico" to the state picture index of the ListView lv_files:

```
//Add state picture
integer index
index = lv_files.AddStatePicture("star.ico")
```

See also

DeleteStatePicture

Arrange

Description

Arranges the icons in rows.

Applies to

ListView controls

Syntax

listviewname.**Arrange** ( )

Argument	Description
listviewname	The name of the ListView control in which you want to arrange icons

Return value

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

Usage

Can only be used with large icon and small icon views.

Examples

This example arranges the icons in a ListView control:

```
lv_list.Arrange()
```

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*)

Argument	Description
mdiframe	The name of an MDI frame window.
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 and 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 and 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 and 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 and 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 **Platform information** 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. If any argument's value is NULL, ArrangeSheets returns NULL.

Examples

This statement in the script for the Clicked event for an item on a menu 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 an item on a menu 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*)

Argument	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*. If *string* is NULL, Asc returns NULL.

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

See also

Char
Mid
Asc in the *DataWindow Reference*

Beep

Description

Causes the computer to beep up to 10 times.

Syntax

Beep (*n*)

Argument	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. If *n* is NULL, Beep returns NULL. The return value usually is not used.

Examples

This statement causes the computer to beep five times:

```
Beep(5)
```

Blob

Description

Converts a string to a blob data type.

Syntax

Blob (*text*)

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

Return value

Blob. Returns the converted string. If *text* is NULL, Blob returns NULL.

Examples

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

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

See also

BlobEdit
BlobMid
String

BlobEdit

Description

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

Syntax

BlobEdit (*blobvariable, n, data*)

Argument	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 returns NULL if there is not enough space in *blobvariable* to copy the data. If any argument's value is NULL, BlobEdit returns NULL.

Examples

This example 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* })

Argument	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. If any argument's value is NULL, BlobMid returns NULL.

Include terminator character
String variables contain a zero terminator, which accounts for one byte. Include the terminator character when calculating how much data to extract.

Examples

In this example, the first call to BlobMid stores 10 bytes of the blob datablob starting at position 5 in the blob data_1; the second call stores 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

BuildModel

Description

Builds either a performance analysis or trace tree model based on the trace file you have specified with the SetTraceFileName function. Optional arguments let you monitor the progress of the build or interrupt it.

You must specify the trace file to be modeled using the SetTraceFileName function before calling BuildModel.

Applies to

Profiling and TraceTree objects

Syntax

instancename.**BuildModel** ({ *progressobject, eventname, triggerpercent* })

Argument	Description
instancename	Instance name of the Profiling or TraceTree object
progressobject (optional)	A PowerObject that represents the number of activities that have been processed
eventname (optional)	A string specifying the name of an event you define
triggerpercent (optional)	A long identifying the number of activities the BuildModel function should process before triggering the *eventname* event

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileNotSetError!—TraceFileName has not been set
- ModelExistsError!—A model has already been built
- EnterpriseOnlyFeature!—This function is supported only in the Enterprise edition of PowerBuilder
- EventNotFoundError!—The event cannot be found on the passed *progressobject*, so the model cannot be built
- EventWrongPrototypeError!—The event was found but does not have the proper prototype, so the model cannot be built
- SourcePBLError!—The source libraries cannot be found, so the model cannot be built

Usage

The BuildModel function extracts raw data from a trace file and maps it to objects that can be acted upon by PowerScript functions. If you want to build a model of your trace file without recording the progress of the build, call BuildModel without any of its optional arguments. If you want to receive progress information while the model is being created or if you want to be able to interrupt a BuildModel that is taking too long to complete, call BuildModel with its optional arguments.

The event *eventname* on the passed *progressobject* is triggered when the number of activities indicated by the *triggerpercent* argument are processed. If the value of *triggerpercent* is 0, *eventname* is triggered for every activity. If the value of *triggerpercent* is greater than 100, *eventname* is never triggered. You define this event using this syntax:

eventname (*currentactivity, totalnumberofactivities*)

Argument	Description
eventname	Name of the event
currentactivity	A long identifying the number of the current activity
totalnumberofactivities	A long identifying the total number of activities in the trace file

Eventname returns a boolean value. If it returns FALSE, the processing initiated by the BuildModel function is canceled and any temporary storage is cleaned up. If you need to stop BuildModel processing that is taking too long, you can return a FALSE value from *eventname*. The script you write for *eventname* determines how progress is monitored. For example, you might display progress or simply check whether the processing must be canceled.

Examples

This example creates a performance analysis model of a trace file:

```
Profiling lpro_model
String ls_filename

lpro_model = CREATE Profiling
lpro_model.SetTraceFileName(ls_filename)
lpro_model.BuildModel()
```

This example creates a trace tree model of a trace file:

```
TraceTree ltct_model
String ls_filename

ltct_model = CREATE TraceTree
ltct_model.SetTraceFileName(ls_filename)
```

```
ltct_model.BuildModel()
```

This example creates a performance analysis model that provides progress information as the model is built. The *eventname* argument to BuildModel is called ue_progress and is triggered each time five percent of the activities have been processed. The progress of the build is shown in a window called w_progress that includes a Cancel button:

```
Profiling lpro_model
String ls_filename
Boolean lb_cancel

lpro_model = CREATE Profiling
lb_cancel = false
lpro_model.SetTraceFileName(ls_filename)

Open(w_progress)
// Call the of_init window function to initialize
// the w_progress window
w_progress.of_init(lpro_model.NumberOfActivities, &
    'Building Model', This, 'ue_cancel')

lpro_model.BuildModel(This, 'ue_progress', 5)

// Clicking the cancel button in w_progress
// sets lb_cancel to true and returns
// false to ue_progress
IF lb_cancel THEN &
    Close(w_progress)
    RETURN -1
END IF
```

See also

SetTraceFileName
DestroyModel

Cancel

Description

Stops the execution of a pipeline object.

Applies to

Pipeline objects

Syntax

pipelineobject.**Cancel** ( )

Argument	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 had reached the maximum errors limit. You control how the pipeline behaves when it reaches the limit in the Data Pipeline painter (see the *PowerBuilder User's Guide*).

Examples

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, SingleLineEdit, RichTextEdit)

Syntax

editname.**CanUndo** ( )

Argument	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, or RichTextEdit 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. If *editname* is NULL, CanUndo returns NULL.

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 and DataStore objects

Syntax

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

Argument	Description
controlname	The name of the graph for which you want the number of categories, or the name of a DataWindow control or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	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 or DataStore

Return value

Integer. Returns the count if it succeeds and -1 if an error occurs. If any argument's value is NULL, CategoryCount returns NULL.

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 and DataStore objects.

Syntax

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

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	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 or DataStore
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 (""). If any argument's value is NULL, CategoryName returns NULL.

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*)

Argument	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*. If *n* is NULL, Ceiling returns NULL.

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 the *DataWindow Reference*

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* })

Argument	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
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 0, which lists the open sheets on the menu bar's next-to-last menu (or the last menu if there is only one available)

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, ChangeMenu returns NULL. 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 on the hidden menu for the frame.

Examples

This statement changes the top-level menu of the w_Employee window to m_Emp1:

```
w_Employee.ChangeMenu(m_Emp1)
```

Char

Description

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

Syntax

Char (*n*)

Argument	Description
n	A string that begins with the character you want, an integer you want to convert to a character, or a blob in which the first value is a string or integer. The rest of the contents of the string or blob is ignored. *N* can also be an Any variable containing a string, integer, or blob

Return value

Char. Returns the first character of *n*. If *n* is NULL, Char returns NULL.

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 the *DataWindow Reference*

Check

Description

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

Applies to

Menu objects

Syntax

menuname.**Check** ( )

Argument	Description
menuname	The fully qualified name of the menu 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. If *menuname* is NULL, Check returns NULL.

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 a menu object's Checked property instead of calling Check.

menuname.Checked = TRUE

This statement:

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

is equivalent to:

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

Examples

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

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

See also

Uncheck

ClassList

Description

Provides a list of the classes included in a performance analysis model.

Applies to

Profiling object

Syntax

instancename.**ClassList** (*list*)

Argument	Description
instancename	Instance name of the Profiling object
list	An unbounded array variable of data type ProfileClass in which ClassList stores a ProfileClass object for each class included in the model. This argument is passed by reference

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- ModelNotExistsError!—The function failed because no model exists

Usage

You use the ClassList function to extract a list of the classes included in a performance analysis model. You must have previously created the performance analysis model from a trace file using the BuildModel function. Each class listed is defined as a ProfileClass object and provides the class name, its parent class and type, and a list of the routines associated with that class. The classes are listed in no particular order.

Examples

This example lists the classes included in the performance analysis model:

```
ProfileClass lproclass_list[], lproclass_class
Profiling lpro_model
Long ll_limitclass, ll_indexclass

lpro_model = CREATE Profiling
lpro_model.BuildModel()

lpro_model.ClassList(lproclass_list)
ll_limitclass = UpperBound(lproclass_list)

FOR ll_indexclass = 1 TO ll_limitclass
   lproclass_class = lproclass_list[ll_indexclass]
...
```

See also

BuildModel

ClassName

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

To determine	Use
The class of an object	Syntax 1
The class (or data type) of a variable	Syntax 2

Syntax 1 For any object

Description

Provides the class (or name) of the specified object.

Applies to

Any control

Syntax

controlname.**Classname** ( )

Argument	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

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

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 ClassName to find 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 descendent 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
```

See also

DraggedObject
TypeOf

Syntax 2

For variables

Description

Provides the data type of a variable.

Syntax

ClassName (*variable*)

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

Return value

String. Returns the name of *variable*. Returns the empty string ("") if *variable* is an enumerated data type or if an error occurs. If *variable* is NULL, ClassName returns NULL.

Usage

ClassName cannot determine the data type if *variable* is an enumerated data type. In this case, ClassName returns the empty string.

Examples

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

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

Clear

Description

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

Platform information
When applied to OLE controls and OLEStorage objects, Clear has no effect on Macintosh and UNIX.

Applies to

MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, DropDownPictureListBox, OLE controls, OLEStorage objects

Syntax

objectname.**Clear** ( )

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

Return value

Long.

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

If *objectname* is NULL, Clear returns NULL.

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 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 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 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 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 a value list or code table associated with a DataWindow column. (A value list 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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**ClearValues** (*column*)

Argument	Description
dwcontrol	The name of a DataWindow control, DataStore, or child DataWindow
column	The column whose value list you want to delete. *Column* can be a column number (integer) or a column name (string). The edit style of the column can be DropDownListBox, Edit, or RadioButton. ClearValues has no effect when *column* has the EditMask or dddw edit style

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, ClearValues returns NULL. The return value is usually not used.

Examples

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

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

See also

GetValue
SetValue

Clipboard

Retrieves or replaces the contents of the system clipboard.

To	Use
Retrieve or replace the contents of the system clipboard with text	Syntax 1
Replace the contents of the system clipboard with a bitmap image of a graph	Syntax 2

Syntax 1

For text

Description

Retrieves or replaces the contents of the system clipboard with text.

Syntax

Clipboard ({ *string* })

Argument	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

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*. If *string* is NULL, Clipboard returns NULL.

Usage

You can use Syntax 1 with the Paste, Replace, or ReplaceText function to insert the clipboard contents in an editable control 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 StaticText st_welcome with the contents of the clipboard:

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

See also

Clear
Copy
Cut
Paste
Replace
ReplaceText

Syntax 2

For bitmaps of graphs

Description

Replaces the contents of the system clipboard with a bitmap image of a graph. You can paste the image into other applications.

Applies to

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

Syntax

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

Argument	Description
controlname	The name of the graph or the DataWindow control or DataStore containing the graph you want to copy to the clipboard
graphcontrol (DataWindow control and DataStore only) (optional)	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, Clipboard returns NULL.

Examples

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")
```

Close

Closes a window, an OLE storage or stream, or a trace file.

To close	Use
A window	Syntax 1
An OLEStorage object variable, saving the object and clearing connections between it and a storage file or object	Syntax 2
A stream associated with the specified OLEStream object variable	Syntax 3
A trace file	Syntax 4

Syntax 1

For windows

Description

Closes a window and releases the storage occupied by the window and all the controls in the window.

Applies to

Window objects

Syntax

Close (*windowname*)

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

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *windowname* is NULL, Close returns NULL. The return value is usually not used.

Usage

Use Syntax 1 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 properties, instance variables, and controls can no longer be referenced in scripts. If a statement in the script references the closed window or its properties or instance variables, an execution error will result.

Preventing a window from closing
You can prevent a window from being closed with a return code of 1 in the script for the CloseQuery event. Use the RETURN statement.

Examples

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

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

After you call Close, the following statements in the script for the CloseQuery event prompt the user for confirmation and prevent the window from closing:

```
IF MessageBox('ExitApplication', &
'Exit?', Question!, YesNo!) = 2 THEN
// If no, stop window from closing
RETURN 1
END IF
```

See also

Hide
Open

Syntax 2

For OLEStorage objects

Description

Closes an OLEStorage object, saving the object in the associated storage file or object and clearing the connection between them. Close also severs connections with any OLE controls that have opened the object. Calling Close is the same as calling Save and then Clear.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLEStorage objects

Syntax

olestorage.**Close** ( )

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

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

If *olestorage* is NULL, Close returns NULL.

Examples

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

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

See also

Open
Save
SaveAs

Syntax 3

For OLEStream objects

Description

Closes an OLEStream object.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLEStream objects

Syntax

olestream.**Close** ( )

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

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

If *olestream* is NULL, Close returns NULL.

Examples

This example closes the OLEStream object stm_pic_label and releases the variable's memory:

```
integer result
result = stm_pic_label.Close()
DESTROY stm_pic_label
```

See also

Open

Syntax 4

For trace files

Description

Closes an open trace file.

Applies to

TraceFile objects

Syntax

instancename.**Close** ( )

Argument	Description
instancename	Instance name of the TraceFile object

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileNotOpenError!—A trace file has not been opened

Usage

You use the Close function to close a trace file you previously opened with the Open function. You use the Close and Open functions as well as the properties and functions of the TraceFile object to access the contents of a trace file directly. You use these functions if you want to perform your own analysis of the tracing data instead of building a model with the Profiling or TraceTree object and the BuildModel function.

Examples

This example closes a trace file:

```
ift_file.Close()
DESTROY ift_file
```

See also

Reset
Open
NextActivity

CloseChannel

Description

Closes a DDE channel.

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

Syntax

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

Argument	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.

Examples

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

CloseTab

Description

Removes a tab page from a Tab control that was opened previously with the OpenTab or OpenTabWithParm function. CloseTab executes the scripts for the user object's Destructor event.

Applies to

Tab controls

Syntax

tabcontrolname.**CloseTab** (*userobjectvar*)

Argument	Description
tabcontrolname	The name of the Tab control containing the tab page you want to close
userobjectvar	The name of the user object you want to close

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, CloseTab returns NULL. The return value is usually not used.

Usage

CloseTab closes a user object that has been opened as a tab page and releases the storage occupied by the object and its controls.

When you call CloseTab, PowerBuilder removes the tab page from the control, 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 CloseTab function.

CloseTab also removes the user object from the Tab control's Control array, which is a property that lists the tab pages within the Tab control. If the closed tab page was not the last element in the array, the index for all subsequent tab pages is reduced by one.

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

Examples

These statements close the tab page user object u_employee and then open the user object u_departments in the Tab control tab_personnel:

```
tab_personnel.CloseTab(u_employee)
tab_personnel.OpenTab(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.CloseTab(u_employee)
   // If user chose No, take no action.
END IF
```

See also OpenTab

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.**CloseUerObject** (*userobjectname*)

Argument	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. If any argument's value is NULL, CloseUserObject returns NULL. 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.

CloseUserObject also removes the user object from the window's Control array, which is a property that lists the window's controls. If the closed user object was not the last element in the array, the index for all subsequent user objects is reduced by one.

After a user object is closed, its properties, instance variables, and controls can no longer be referenced in scripts. If a statement in the script references the closed user object or its properties 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*)

Argument	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. If any argument's value is NULL, CloseWithReturn returns NULL. 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 CloseWithReturn function.

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

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

Return value data type	Message object property
Numeric	Message.DoubleParm

Return value data type	Message object property
PowerObject (such as 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 property 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 properties of that control.

Preventing a window from closing
You can prevent a window from being closed with a return code of 1 in the script for the CloseQuery event. Use a RETURN statement.

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 for the user's experience level. The user types an experience level in the SingleLineEdit control sle_signon_id. The OK button in the response window passes the text in sle_signon_id back to the menu item script. The menu item script checks the StringParm property 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 menu item 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

CollapseItem

Description

Collapses the specified item.

Applies to

TreeView controls

Syntax

treeviewname.**CollapseItem** (*itemhandle*)

Argument	Description
treeviewname	The TreeView control in which you want to collapse an item
itemhandle	The handle of the item you want to collapse

Return value

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

Usage

To collapse an entire tree, use the RootTreeItem handle as the argument.

Examples

This example collapses an item in a TreeView control:

```
long ll_tvi
ll_tvi = tv_list.FindItem(currenttreeitem!, 0)
tv_list.CollapseItem(ll_tvi)
```

This example collapses all items in a TreeView control:

```
long ll_tvi
ll_tvi = tv_list.FindItem(roottreeitem!, 0)
tv_list.CollapseItem(ll_tvi)
```

See also

ExpandItem
ExpandAll

CommandParm

Description

Retrieves the argument 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 argument string if it succeeds and the empty string ("") if it fails or if there were no arguments.

Usage

Command arguments 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>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 arguments, CommandParm returns them all as a single string. You can use string functions, such as Mid and Pos, to parse the string.

You don't need to call CommandParm in the application's Open event. Use the argument instead.

Examples

These statements retrieve the command line arguments and save them in the variable ls_command_line:

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

If the command line holds several arguments, you can use string functions to separate the arguments. This example stores a variable number of arguments, obtained with CommandParm, in an array. The code assumes each argument is separated by one space.For each argument, the Pos function searches for a space; the Left function copies the argument to the array; and Replace removes the argument from the original string so the next argument moves to the first position:

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

```
// Get the arguments 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*)

Argument	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 property of an OLE 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 property of a control
-2 Class name not found
-3 Object could not be created
-4 Could not connect to object
-9 Other error

If any argument's value is NULL, ConnectToNewObject returns NULL.

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 INFO For more information about OLE automation, see ConnectToObject.

Examples

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

ConnectToNewRemoteObject

Description

Creates a new OLE object in the specified remote server application (if security on the server allows it) and associates the new object with a PowerBuilder OLEObject variable. ConnectToNewRemoteObject starts the server application if necessary.

Applies to

OLEObject objects

Syntax

oleobject.**ConnectToNewRemoteObject** (*hostname*, *classname*)

Argument	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 property of an OLE control
hostname	A string whose value is the name of the remote host where the COM server is located
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 property of a control
-2 Class name not found
-3 Object could not be created
-4 Could not connect to object
-9 Other error
-10 Feature not supported on this platform
-11 Server name is invalid
-12 Server does not support operation
-13 Access to remote host denied
-14 Server unavailable

Usage

The OLEObject variable is used for OLE automation, in which the PowerBuilder application asks the server application to manipulate the OLE object programmatically. ConnectToNewRemoteObject can only be used with servers that support remote activation.

FOR INFO For more information about OLE automation, see ConnectToObject. For information about connecting to objects on a remote host, see ConnectToRemoteObject.

Examples

This example creates an OLEObject variable and calls ConnectToNewRemoteObject to create and connect to a new Excel object on a remote host named ulysses:

```
integer result
OLEObject myoleobject

myoleobject = CREATE OLEObject
result = myoleobject.ConnectToNewRemoteObject( &
    "ulysses", "Excel.application")
```

See also

ConnectToObject
ConnectToRemoteObject

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* })

Argument	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 property of an OLE 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 property 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

If any argument's value is NULL, ConnectToObject returns NULL.

Usage

After you have created an OLEObject variable and connected it to an OLE object and its server application, you can set properties 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 property names are invalid, an execution 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 properties 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 properties for an OLE object in an OLE control. To do so, specify the Object property 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 control ole_1:

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

Examples

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.workbooks(1).&
```

```
        worksheets(1).cells(1,1).value = 14
    myoleobject.application.workbooks(1).save()
END IF
DESTROY myoleobject
```

This example connects to an Excel chart (using a Windows as pathname):

```
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

ConnectToRemoteObject

Description

Associates an OLE object with a PowerBuilder OLEObject variable and starts the server application.

Applies to

OLEObject objects

Syntax

oleobject.**ConnectToRemoteObject** (*hostname*, *filename* {, *classname* })

Argument	Description
oleobject	The name of an OLEObject variable that you want to connect to an OLE object. You cannot specify an OLEObject that is the Object property of an OLE control
hostname	A string whose value is the name of the remote host where the COM server is located
filename	A string whose value is the name of an OLE storage file. You cannot specify an empty string. COM looks for *filename* on the local (client) machine. If *filename* is located on the remote host, its location must be made available to the local host by sharing. Use the share name for the remote drive to specify a file on a remote host — for example, \\hostname\shared_directory\test.ext
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* and *filename*, is an OLE structured storage file, PowerBuilder uses the class ID in *filename*. Otherwise, PowerBuilder uses the filename extension 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 property of a control
-2 Class name not found
-3 Object could not be created
-4 Could not connect to object
-5 Could not connect to the currently active object
-6 File name is invalid
-7 File not found or could not be opened
-8 Load from file not supported by server
-9 Other error
-10 Feature not supported on this platform
-11 Server name is invalid
-12 Server does not support operation

-13 Access to remote host denied
-14 Server unavailable

Usage

The OLEObject variable is used for OLE automation, in which the PowerBuilder application asks the server application to manipulate the OLE object programmatically. ConnectToRemoteObject can only be used with servers that support remote activation.

The following information applies to creating or instantiating and binding to OLE objects on remote hosts.

FOR INFO For general information about OLE automation, see ConnectToObject.

DCOM support ConnectToRemoteObject and ConnectToNewRemoteObject use DCOM (Distributed Component Object Model). Support for DCOM is not available on all platforms. It is available now on Windows NT 4.*x* and on Windows 95 with the DCOM service pack. DCOM on Windows 95 does not currently support secure remote activation. Windows 95 can therefore be used for DCOM clients but not for DCOM servers.

Security Security on the server must be configured correctly to launch objects on remote hosts. Security is configured using registry keys. You must specify attributes for allowing and disallowing launching of servers and connections to running objects to allow client access. You can update the registry manually or with a tool such as DCOMCNFG.EXE or OLE Viewer.

Registry entries The server application must be registered on both the server and the client.

To find files other than OLE structured storage files, registry entries must include a file extension entry, such as .xls for Excel. If the file is a structured storage file, then COM reads the file and extracts the server identity from the file; otherwise, the registry entry for the file extension is used and the appropriate server application is launched.

If the DCOM server uses a custom interface, the proxy/stub DLL for the interface must be registered on the client. The proxy/stub DLL is created by the designer of the custom interface. It handles the marshaling of parameters through the proxy on the client and the stub on the server so that a remote procedure call can take place.

Examples

This example declares and creates an OLEObject variable and connects to an Excel worksheet on a remote host named falco. The drive where the worksheet resides is mapped as f:\excel on the local host:

```
integer result
OLEObject myoleobject

myoleobject = CREATE OLEObject
result = myoleobject.ConnectToRemoteObject( &
    "falco", "f:\excel\expense.xls")
```

This example connects to the same object on the remote host but opens it as an Excel chart:

```
integer result
OLEObject myoleobject

myoleobject = CREATE OLEObject
result = myoleobject.ConnectToRemoteObject( &
    "falco", "f:\excel\expense.xls", "Excel.chart")
```

See also

ConnectToNewRemoteObject
ConnectToObject
DisconnectObject

ConnectToServer

Description

Connects a client application to a server application. The client application must call ConnectToServer before it can use a remote object defined on the server application.

This function applies to distributed applications only.

Applies to

Connection objects

Syntax

connection.**ConnectToServer** ( )

Argument	Description
connection	The name of the Connection object you want to use to establish the connection. The Connection object has properties that specify how the connection will be established

Return value

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

50 Distributed service error
52 Distributed communications error
53 Requested server not active
54 Server not accepting requests
55 Request terminated abnormally
56 Response to request incomplete
62 Server busy

Usage

Before calling ConnectToServer, you assign values to the properties of the Connection object. The properties you set vary depending on which communications driver you are using.

Examples

In this example, the client application connects to a server application using the Connection object myconnect. The communications driver used for the connection is WinSock:

```
connection myconnect
myconnect = create connection
myconnect.driver = "WinSock"
myconnect.application = "dpbserv"
myconnect.location = "server01"
myconnect.ConnectToServer()
```

See also

DisconnectServer

Copy

Description

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

Platform information
When applied to OLE controls and objects, Copy has no effect on Macintosh and UNIX.

Applies to

DataWindow, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, DropDownPictureListBox, OLE controls, and OLE DWObjects (objects within a DataWindow object that is within a DataWindow control)

Syntax

objectref.**Copy** ( )

Argument	Description
objectref	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox or DropDownPictureListBox containing the text you want to copy to the clipboard. *or* The name of the OLE control or the fully qualified name of a OLE DWObject within a DataWindow control that contains the object you want to copy to the clipboard. The fully qualified name for a DWObject has this syntax: *dwcontrol*.**Object**.*dwobjectname* If *objectref* is a DataWindow, text is copied from the edit control over the current row and column. If *objectref* is a DropDownListBox or DropDownPictureListBox, its AllowEdit property must be TRUE

Return value

Long.

For RichTextEdit controls, Copy returns a long. For other edit controls and OLE objects, Copy returns an integer.

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

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

-1 Container is empty
-2 Copy Failed
-9 Other error

If *objectref* is NULL, Copy returns NULL.

Usage

To select text for copying, the user can use the mouse or keyboard. You can also call the SelectText function in a script. For RichTextEdit controls, there are several additional functions for selecting text: SelectTextAll, SelectTextLine, and SelectTextWord.

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 control ole_1 to the clipboard:

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

See also

Clear
Clipboard
Cut
Paste
ReplaceText
SelectText

CopyRTF

Description

Returns the selected text, pictures, and input fields in a RichTextEdit control or RichText DataWindow as a string with rich text formatting. Bitmaps and input fields are included in the string.

Applies to

DataWindow controls, DataStore objects, and RichTextEdit controls

Syntax

rtename.**CopyRTF** ({ *selected* {, *band* } })

Argument	Description
rtename	The name of the DataWindow control, DataStore object, or RichTextEdit control from which you want to copy the selection in rich text format. The DataWindow object in the DataWindow control or DataStore must be a RichText DataWindow
selected (optional)	A boolean value indicated whether to copy selected text only. Values are: ♦ TRUE — (Default) Copy selected text only ♦ FALSE — Copy the entire contents of the band
band (optional)	A value of the Band enumerated data type specifying the band from which to copy text. Values are: ♦ Detail! — Copy text from the detail band ♦ Header! — Copy text from the header band ♦ Footer! — Copy text from the footer band The default is the band that contains the insertion point

Return value

String. Returns the selected text as a string.

CopyRTF returns an empty string ("") if:

- There is no selection and *selected* is TRUE
- An error occurs

Usage

CopyRTF does not involve the clipboard. The copied information is stored in a string. If you use the standard clipboard functions (Copy and Cut) the clipboard will contain the text without any formatting.

To incorporate the text with RTF formatting into another RichTextEdit control, use PasteRTF.

FOR INFO For more information about rich text format, see the chapter about implementing rich text in *Application Techniques*.

Examples

This statement returns the text that is selected in the RichTextEdit rte_message and stores it in the string ls_richtext:

```
string ls_richtext
ls_richtext = rte_message.CopyRTF()
```

This example copies the text in rte_1, saving it in ls_richtext, and pastes it into rte_2. The user clicks the RadioButton rb_true to copy selected text and rb_false to copy all the text. The number of characters pasted is saved in ll_numchars reported in the StaticText st_status:

```
string ls_richtext
boolean lb_selected
long ll_numchars

IF rb_true.Checked = TRUE THEN
   lb_selected = TRUE
ELSE
   lb_selected = FALSE
END IF

ls_richtext = rte_1.CopyRTF(lb_selected)
ll_numchars = rte_2.PasteRTF(ls_richtext)
st_status.Text = String(ll_numchars)
```

See also

PasteRTF

Cos

Description

Calculates the cosine of an angle.

Syntax

Cos (*n*)

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

Return value

Double. Returns the cosine of *n*. If *n* is NULL, Cos returns NULL.

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 the *DataWindow Reference*

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.

Examples

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 ll_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 or DataStore object. This dynamic DataWindow object does not become a permanent part of the application source library.

Applies to

DataWindow controls and DataStore objects

Syntax

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

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore 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. If any argument's value is NULL, Create returns NULL.

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 *dwcontrol*.

DataWindow source code syntax is complex. You can use the Describe and LibraryExport functions to obtain the source code of existing DataWindows to use as models. You can also use the Library painter and export the syntax of a DataWindow object. Another source of DataWindow code is the SyntaxFromSQL function, which creates DataWindow source code based on an SQL statement. Many values in the source code syntax correspond to properties of the DataWindow object, which are documented in the *DataWindow Reference*.

When you examine syntax for existing DataWindow objects, you will see that the order of the syntax can vary. Release must be the first statement, and DataWindow should be the next statement. If you change the order, use care; the order can affect the results.

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

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.

  When you use this 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.

- 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 this method, the comment can be all or part of a line or multiple lines; the compiler ignores everything between /* and */.

Examples

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

CreateInstance

Description

Creates an instance of a remote object.

Applies to

Connection objects

Syntax

connection.**CreateInstance** (*objectvariable* {, *classname* })

Argument	Description
connection	The name of the Connection object used to establish the connection
objectvariable	A global, instance, or local variable whose data type is the same class as the object being created or an ancestor of that class
classname (optional)	A string whose value is the name of the class data type to be created

Return value

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

50 Distributed service error
52 Distributed communications error
53 Requested server not active
54 Server not accepting requests
55 Request terminated abnormally
56 Response to request incomplete
62 Server busy

Usage

Before calling CreateInstance, you need to connect to a server. To do this, you need to call the ConnectToServer function.

CreateInstance allows you to create an object on a remote server. If you want to create an object locally, you need to use the CREATE statement.

When you deploy a remote object's class definition in a client application, the definition on the client has the same name as the remote object definition deployed in the server application. Variables declared with this type are able to hold a reference to a local object or a remote object. Therefore, at execution time, you can instantiate the object locally (with the CREATE statement) or remotely (with the CreateInstance function) depending on your application requirements. In either case, once you have created the object, its physical location is transparent to client-side scripts that use the object.

Obsolete function
CreateInstance replaces the SetConnect function. SetConnect is obsolete and will be discontinued in the near future. You should replace all use of SetConnect with the CreateInstance function as soon as possible.

Examples

The following statements create an object locally or remotely depending on the outcome of a test. The statements use the CreateInstance function to create a remote object and the CREATE statement to create a local object:

```
boolean bWantRemote
connection myconnect
uo_customer iuo_customer

//Determine whether you want a remote
//object or a local object.
...
//Then create the object.
IF bWantRemote THEN
   //Create a remote object
   IF myconnect.CreateInstance(iuo_customer) <> 0 THEN
      //deal with the error
   ...
   END IF
ELSE
   //Create a local object
   iuo_customer = CREATE uo_customer
END IF

//Call a function of the object.
//The function call is the same whether the object was
//created on the server or the client.
IF isValid(iou_customer) THEN
   iuo_customer.GetCustomerData()
END IF
```

See also

ConnectToServer

CreatePage

Description

Creates a tab page if it has not already been created.

Applies to

User objects used as tab pages

Syntax

userobject.**CreatePage** ( )

Argument	Description
userobject	The name of the tab page you want to create

Return value

Integer. Returns one of the following values:1 if the page is successfully created and -1 if the page was already created or if it's not a tab page.

1 — The tab page was successfully created
0 — The tab page has already been created
-1 — The user object is not a tab page

Usage

A window will open more quickly if the creation of graphical representations is delayed for tab pages with many controls. However, scripts cannot refer to a control on a tab page until the control's Constructor event has run and a graphical representation of the control has been created. When the CreateOnDemand property of the Tab control is selected, scripts cannot reference controls on tab pages that the user hasn't viewed. CreatePage allows you to create a tab page if it has not already been created.

Examples

This example tests whether tabpage_2 has been created and, if not, creates it:

```
IF tab_1.CreateOnDemand = True THEN
   IF tab_1.tabpage_2.PageCreated() = False THEN
      tab_1.tabpage_2.CreatePage()
   END IF
END IF
```

See also

PageCreated

CrosstabDialog

Description

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

Applies to

DataWindow controls

Syntax

dwcontrol.**CrossTabDialog** ( )

Argument	Description
dwcontrol	The name of the DataWindow control for which you want to display the Crosstab Definition dialog box

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *dwcontrol* is NULL, CrosstabDialog returns NULL.

Usage

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

Examples

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, replacing the clipboard contents with the deleted text or object.

Platform information
When applied to OLE controls, Cut has no effect on Macintosh and UNIX.

Applies to

DataWindow, MultiLineEdit, SingleLineEdit, DropDownListBox, DropDownPictureListBox, and OLE controls

Syntax

controlname.**Cut** ( )

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

Return value

Long.

For editable controls, Cut 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 controls, Cut returns 0 if it succeeds and one of the following negative values if an error occurs:

-1 Container is empty
-2 Cut failed
-9 Other error

If *controlname* is NULL, Cut returns NULL.

Usage

To select text for deleting, the user can use the mouse or keyboard. You can also call the SelectText function in a script. For RichTextEdit controls, there are several additional functions for selecting text: SelectTextAll, SelectTextLine, and SelectTextWord.

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.

Examples

Assuming 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 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 and DataStore objects

Syntax

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

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control or DataStore only) (optional)	The name of the graph in the DataWindow control or DataStore 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. If any argument's value is NULL, DataCount returns NULL.

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

DataSource

Description

Allows a RichTextEdit control to share data with a DataWindow and display the data in its input fields. If there are input fields in the RichTextEdit control that match the names of columns in the DataWindow, the data in the DataWindow is assigned to those input fields. The document in the RichTextEdit control is repeated so that there is an instance of the document for each row in the DataWindow.

Applies to

RichTextEdit controls

Syntax

rtename.**DataSource** (*dwsource*)

Argument	Description
rtename	The name of the RichTextEdit control for which you want to get data in a DataWindow
dwsource	The name of the DataWindow control, DataStore, or child DataWindow that contains the data to be connected with input fields in *rtename*

Return value

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

Usage

When names of input fields match names of columns in the DataWindow object, the data in the columns is assigned to the matching input fields.

The document in the RichTextEdit control is associated with one row in the DataWindow. There is an instance of the document for each retrieved row. The text in the RichTextEdit control is repeated, with all its formatting, in every document instance. The content of the input fields changes as the data in each row changes. Except for the contents of the input fields, the contents of each instance is the same—you cannot make changes to the surrounding text that affect individual instances only.

If the InputFieldNamesVisible property of the RichTextEdit control is TRUE, the fields will show their names instead of the data they contain. Change the property value to FALSE to see the data.

The following RichTextEdit functions operate on or report information about an instance of the document:

LineCount
PageCount
InsertDocument
SaveDocument
SelectedPage
SelectedStart

SelectedLine
SelectText
SelectTextAll

The following RichTextEdit function affects the collection of documents:

Print

Examples

This example establishes the DataWindow control dw_1 as the data source for the RichTextEdit rte_1:

```
rte_1.DataSource(dw_1)
```

This example inserts a document called LETTER.RTF into the RichTextEdit rte_letter (the names of the document's input fields match the columns in a DataWindow object d_emp), creates a DataStore, associates it with d_emp, and retrieves data. Then it inserts the document in rte_letter and sets up the DataStore as the data source for rte_1:

```
DataStore ds_empinfo
ds_empinfo = CREATE DataStore
ds_empinfo.DataObject = "d_emp"
ds_empinfo.SetTransObject(SQLCA)
ds_empinfo.Retrieve()

rte_letter.InsertDocument("LETTER.RTF", TRUE)
rte_letter.DataSource(ds_empinfo)
```

See also

InputFieldChangeData
InputFieldCurrentName
InputFieldDeleteCurrent
InputFieldGetData
InputFieldInsert

Date

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

To	Use
Extract the date from DateTime data or extract a date stored in a blob	Syntax 1
Convert a string to a date	Syntax 2
Combine numeric data into a date	Syntax 3

Syntax 1

For DateTime data and blobs

Description

Extracts a date from a DateTime value or from a blob whose first value is a date or DateTime value.

Syntax

Date (*datetime*)

Argument	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. *Datetime* can also be an Any variable containing a DateTime or blob

Return value

Date. Returns the date in *datetime* as a date. If *datetime* contains an invalid date or an incompatible data type, Date returns 1900-01-01. If *datetime* is NULL, Date returns NULL.

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)
```

Assuming 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))
```

See also

DateTime

Syntax 2

For strings

Description

Converts a string whose value is a valid date to a date value.

Syntax

Date (*string*)

Argument	Description
string	A string containing a valid date (such as January 1, 1998, or 12-31-99) that you want returned as a date. *Datetime* can also be an Any variable containing a string

Return value

Date. Returns the date in *string* as a date. If *string* contains an invalid date or an incompatible data type, Date returns 1900-01-01. If *string* is NULL, Date returns NULL.

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). PowerBuilder assumes a 4-digit number is a year. Leading zeros are optional for month and day. The month, whether a name, an abbreviation, or a number, must be in the month location specified in the system setting for a date's format. If you do not know the system setting, use the standard data type date format yyyy-mm-dd.

Date literals do not need to be converted with the Date function.

Examples

These statements all return the date data type for text expressing the date July 4, 1994 (1994-07-04). The system setting for a date's format is set with the month's position in the middle:

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

Date("1994 July 4")

Date("04 July 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
```

See also

DateTime
IsDate
RelativeDate
RelativeTime
Date in the *DataWindow Reference*

Syntax 3 For combining numbers into a date

Description

Combines numbers representing the year, month, and day into a date value.

Syntax

Date (*year, month, day*)

Argument	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

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

Examples

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
DaysAfter
RelativeDate
RelativeTime

DateTime

Manipulates DateTime values. There are two syntaxes.

To	Use
Combine a date and a time value into a DateTime value	Syntax 1
Obtain a DateTime value that is stored in a blob	Syntax 2

Syntax 1

For creating DateTime values

Description

Combines a date value and a time value into a DateTime value.

Syntax

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

Argument	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

DateTime. Returns a DateTime value based on the values in *date* and optionally *time*. If any argument's value is NULL, DateTime returns NULL.

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)
```

See also

Date
Time

DateTime in the *DataWindow Reference*

Syntax 2 For extracting DateTime values from blobs

Description

Extracts a DateTime value from a blob.

Syntax

DateTime (*blob*)

Argument	Description
blob	A blob in which the first value is a DateTime value. The rest of the contents of the blob is ignored. *Blob* can also be an Any variable containing a blob

Return value

DateTime. Returns the DateTime value stored in *blob*. If *blob* is NULL, DateTime returns NULL.

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

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

Day

Description

Obtains the day of the month in a date value.

Syntax

Day (*date*)

Argument	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*. If *date* is NULL, Day returns NULL.

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 the *DataWindow Reference*

DayName

Description

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

Syntax

DayName (*date*)

Argument	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*. If *date* is NULL, DayName returns NULL.

Usage

DayName returns a name in the language of the deployment kit (DDDK) available on the machine where the application is run. If you have installed a localized deployment kit (DDDK) in the development environment or on a user's machine, then on that machine the name returned by DayName will be in the language of the localized DDDK.

FOR INFO For information about the localized deployment kits, which are available in French, German, Italian, Spanish, Dutch, Danish, Norwegian, and Swedish, see the *PowerBuilder User's Guide*.

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 the *DataWindow Reference*

DayNumber

Description

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

Syntax

DayNumber (*date*)

Argument	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. If *date* is NULL, DayNumber returns NULL.

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 the *DataWindow Reference*

DaysAfter

Description

Determines the number of days one date occurs after another.

Syntax

DaysAfter (*date1*, *date2*)

Argument	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. If any argument's value is NULL, DaysAfter returns NULL.

Examples

This statement returns 4:

```
DaysAfter(1996-12-20, 1996-12-24)
```

This statement returns -4:

```
DaysAfter(1996-12-24, 1996-12-20)
```

This statement returns 0:

```
DaysAfter(1996-12-24, 1996-12-24)
```

This statement returns 5:

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

If you declare date1 and date2 date variables and assign February 16, 1996, to date1 and April 28, 1996, to date2 as follows:

```
date date1, date2

date1 = 1996-02-16
date2 = 1996-04-28
```

then each of the following statements returns 71:

```
DaysAfter(date1, date2)

DaysAfter(1996-02-16, date2)

DaysAfter(date1, 1996-04-28)

DaysAfter(1996-02-16, 1996-04-28)
```

See also

RelativeDate
RelativeTime
SecondsAfter
DaysAfter in the *DataWindow Reference*

DBCancel

Description

Cancels the retrieval in process in a DataWindow.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**DBCancel** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If *dwcontrol* is NULL, DBCancel returns NULL.

Usage

To cancel a database retrieval, you need two pieces of code:

- A script that calls DBCancel. To let the user cancel the retrieval, you could call it (or a user function that calls it) in a script for an item on a menu or CommandButton. This script would generally set an instance variable to indicate that the user requested cancellation:

```
ib_cancel = TRUE
dw_1.DBCancel()
```

- Code in the script for the RetrieveRow event that returns a value of 1 to stop the retrieval:

```
IF ib_cancel = TRUE THEN
   RETURN 1
END IF
```

Coding something in the RetrieveRow event's script (even just a comment) enables the operating system to process events while the DataWindow is being populated with rows from the database. If the RetrieveRow event's script is empty, menus and command buttons can't even be clicked until the retrieval is completely finished. This can be frustrating if the user inadvertently starts a retrieval that is going to take a long time.

If the Async DBParm parameter is set to 1 (for asynchronous operation), a user or a script can cancel a query either before the first row is returned or during the data retrieval process. If Async is set to 0 (for synchronous operation), the user cannot select the menu or CommandButton until the first row is retrieved. The asynchronous setting is useful when a query may take a long time to retrieve its first row.

FOR INFO For a list of the DBMSs that support the Async DBParm parameter, see *Connecting to Your Database*.

Examples

In this example, the menu bar for an MDI application has menu items for starting and canceling a retrieval. When the user cancels the retrieval, a user function calls DBCancel and sets a boolean instance variable to TRUE. The RetrieveStart and RetrieveRow events check this variable and return the appropriate value.

In this hypothetical application, the user starts a retrieval by selecting Retrieve from a menu. The script for the Retrieve menu item calls a user function for the window:

```
w_async1.wf_retrieve()
```

The wf_retrieve function sets the Async DBParm for asynchronous processing and starts the retrieval. Because Async is set to 1, the user can select the Cancel menu item at any time, even before the first row is retrieved (in your own application, you would include error handling to make sure Retrieve returned successfully):

```
long rc
ib_cancel = FALSE
SQLCA.DBParm = 'Async = 1'
rc = dw_1.Retrieve()
```

The user can stop the retrieval by selecting Cancel from the menu. The script for the Cancel menu item reads:

```
w_async1.wf_cancel()
```

The user function wf_cancel for the window w_async1 calls DBCancel and sets a flag indicating that the retrieval is canceled. Other events for the DataWindow will check this flag and abort the retrieval too. The variable ib_cancel is an instance variable for the window:

```
ib_cancel = TRUE
dw_1.DBCancel()
```

Scripts for the RetrieveStart and RetrieveRow events both check the ib_cancel instance variable and, if it is TRUE, stop the retrieval by returning a value of 1. In order to cancel the retrieval, some code or comment in the script for the RetrieveRow event is required:

```
IF ib_cancel = TRUE THEN
   RETURN 1
END IF
```

See also

Retrieve

DBErrorCode

Description

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

Obsolete function
DBErrorCode is obsolete and will be discontinued in the near future. You should replace all use of DBErrorCode as soon as possible. The database error code is available as an argument in the DBError event.

Applies to

DataWindow controls and child DataWindows

Syntax

dwcontrol.**DBErrorCode** ( )

Argument	Description
dwcontrol	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 an error code when a database error occurs in *dwcontrol*. Error codes -1 through -4 are PowerBuilder codes. Other codes are database-specific. Returns 0 if there is no error. If *dwcontrol* is NULL, DBErrorCode returns NULL.

PowerBuilder error codes are:

- -1 Can't connect to the database because of missing values in the transaction object.
- -2 Can't connect to the database.
- -3 The key specified in an Update or Retrieve no longer matches an existing row. (This can happen when another user has changed the row after you retrieved it.)
- -4 Writing a blob to the database failed.

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. A return code of 1 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
   RETURN 1
END IF
```

See also

DBErrorMessage
MessageBox

DBErrorMessage

Description

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

Obsolete function
DBErrorMessage is obsolete and will be discontinued in the near future. You should replace all use of DBErrorMessage as soon as possible. The database error message is available as an argument in the DBError event.

Applies to

DataWindow controls and child DataWindows

Syntax

dwcontrol.**DBErrorMessage** ( )

Argument	Description
dwcontrol	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 *dwcontrol*. Returns the empty string ("") if there is no error. If *dwcontrol* is NULL, DBErrorMessage returns NULL.

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.DBErrorMessage()
```

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
   RETURN 1
END IF
```

See also

DBErrorCode
MessageBox

DBHandle

Description

Reports the handle for your DBMS.

Applies to

Transaction objects

Syntax

transactionobject.**DBHandle** ( )

Argument	Description
transactionobject	The current transaction object

Return value

UnsignedLong. Returns the handle for your DBMS. *Transactionobject* must exist, and the database must be connected. If *transactionobject* is NULL, DBHandle returns NULL. If *transactionobject* does not exist, an execution error occurs. If there is not enough memory to connect to your DBMS, DBHandle returns a negative number.

Usage

DBHandle returns a valid handle only if you are connected to the database. It is not able to determine if the database connection does not exist or has been lost.

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.

DebugBreak

Description	Suspends execution and opens the Debug window.
Syntax	**DebugBreak** ( )
Return value	None
Usage	Insert a call to the DebugBreak function into a script at a point at which you want to suspend execution and examine the application. Then enable just-in-time debugging and run the application in the development environment. When PowerBuilder encounters the DebugBreak function, the Debug window opens showing the current context.
Examples	This statement tests whether a variable is NULL and opens the Debug window if it is: `IF IsNull(auo_ext) THEN DebugBreak()`

Dec

Description

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

Syntax

Dec (*stringorblob*)

Argument	Description
stringorblob	A 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. *Stringorblob* can also be an Any variable containing a string or blob

Return value

Decimal. Returns the value of *stringorblob* as a decimal. If *stringorblob* is not a valid PowerScript number or if it contains an incompatible data type, Dec returns 0. If *stringorblob* is NULL, Dec returns NULL.

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.

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*)

Argument	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. If any argument's value is NULL, DeleteCategory returns NULL.

Examples

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

DeleteColumn

Description

Deletes a column.

ListView controls

Syntax

listviewname.**DeleteColumn** (*index*)

Argument	Description
listviewname	The name of the ListView control from which you want to delete a column
index	The index number of the column you want to delete

Return value

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

Examples

This example deletes the second column in a ListView control:

```
lv_list.DeleteColumn(2)
```

See also

DeleteColumns

DeleteColumns

Description

Deletes all columns.

Applies to

ListView controls

Syntax

listviewname.**DeleteColumns ( )**

Argument	Description
listviewname	The name of the ListView control from which you want to delete all columns

Return value

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

Examples

This example deletes all columns in a ListView control:

```
lv_list.DeleteColumns()
```

See also

DeleteColumn

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*)

Argument	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. If any argument's value is NULL, DeleteData returns NULL.

Examples

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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**DeletedCount** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol* but not updated in the associated database table.

Returns 0 if no rows have been deleted or if all the deleted rows have been updated in the database table. DeletedCount returns -1 if it fails. If any argument's value is NULL, DeletedCount returns NULL.

Usage

An updatable DataWindow control or DataStore 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.

If a DataWindow is not updatable, rows that are deleted are discarded—they are not stored in the delete buffer. Therefore, DeletedCount always returns 0 for a nonupdatable DataWindow.

Examples

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

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

This 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

Deletes an item from a ListBox, DropDownListBox, or ListView control.

To delete an item from	Use
A ListBox or DropDownListBox control	Syntax 1
A ListView control	Syntax 2
A TreeView control	Syntax 3

Syntax 1

For ListBox and DropDownListBox controls

Description

Deletes an item from the list of values for a listbox control.

Applies to

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

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

Argument	Description
listboxname	The name of the ListBox, DropDownListBox, PictureListBox, or DropDownPictureListBox from 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. If any argument's value is NULL, DeleteItem returns NULL.

Usage

If the control's Sorted property 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

Assuming 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)
```

This statement deletes the item "Personal" from the ListBox lb_purpose:

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

See also

AddItem
FindItem
InsertItem
SelectItem

Syntax 2

For ListView controls

Description

Deletes the specified item from a ListView control.

Applies to

ListView controls

Syntax

listviewname.**DeleteItem** (*index*)

Argument	Description
listviewname	The name of the ListView control from which you want to delete an item
index	The index number of the item you want to delete

Return value

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

Examples

This example uses SelectedIndex to find the index of the selected ListView item and then deletes the corresponding item:

```
integer index

index = lv_list.selectedindex()
lv_list.DeleteItem(index)
```

See also

AddItem
FindItem
InsertItem

SelectItem
DeleteItems

Syntax 3 For TreeView controls

Description

Deletes an item from a control and all its child items, if any.

Applies to

TreeView controls

Syntax

treeviewname.**DeleteItem** (*itemhandle*)

Argument	Description
treeviewname	The name of the TreeView control from which you want to delete an item
itemhandle	The handle of the item you want to delete

Return value

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

Usage

If all items are children of a single item at the root level, you can delete all items in the TreeView with the handle for RootTreeItem as the argument for DeleteItem. Otherwise, you need to loop through the items at the first level.

Examples

This example deletes an item from a TreeView control:

```
long ll_tvi
ll_tvi = tv_list.FindItem(CurrentTreeItem!, 0)
tv_list.DeleteItem(ll_tvi)
```

This example deletes all items from a TreeView control when there are several items at the first level:

```
long tvi_hdl = 0
DO UNTIL tv_1.FindItem(RootTreeItem!, 0) = -1
   tv_1.DeleteItem(tvi_hdl)
LOOP
```

See also

AddItem
FindItem
InsertItem
SelectItem
DeleteItems

DeleteItems

Description

Deletes all items from a ListView control.

Applies to

ListView controls

Syntax

listviewname.**DeleteItems** ( )

Argument	Description
listviewname	The name of the ListView control from which you want to delete all items

Return value

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

Examples

This example deletes all the items in a ListView control:

```
lv_list.DeleteItems()
```

See also

DeleteItem

DeleteLargePicture

Description

Deletes a picture from the large image list.

Applies to

ListView controls

Syntax

listviewname.**DeleteLargePicture** (*index*)

Argument	Description
listviewname	The name of the ListView control to which you want to delete a large picture from the image list
index	The index entry for the large picture you want to delete

Return value

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

Examples

This example deletes a large picture from a ListView control:

```
lv_list.DeleteLargePicture(1)
```

See also

DeleteLargePictures

DeleteLargePictures

Description

Deletes all large pictures from a ListView control.

Applies to

ListView controls

Syntax

listviewname.**DeleteLargePictures** ( )

Argument	Description
listviewname	The name of the ListView control from which you want to delete all pictures from the large picture image list

Return value

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

Examples

This example deletes all large pictures from a ListView control:

```
lv_list.DeleteLargePictures()
```

See also

DeleteLargePicture

DeletePicture

Description

Deletes a picture from the image list.

Applies to

PictureListBox, DropDownPictureListBox, and TreeView controls

Syntax

controlname.**DeletePicture** (*index*)

Argument	Description
controlname	The control from which you want to delete a picture
index	The index number of the picture you want to delete from the TreeView control's image list

Return value

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

Usage

When you delete a picture from the image list for a control, all subsequent pictures in the list are renumbered to fill the gap. Because the picture index for an item doesn't change, the pictures for items that use the affected index numbers will change.

Examples

This example deletes the sixth image from the image list:

```
tv_list.DeletePicture(6)
```

See also

DeletePictures
AddPicture

DeletePictures

Description

Deletes all pictures from an image list.

Applies to

PictureListBox, DropDownPictureListBox, and TreeView controls

Syntax

controlname.**DeletePictures** ( )

Argument	Description
controlname	The control in which you want to delete all pictures from the image list

Return value

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

Examples

This example deletes all images from a TreeView control image list:

```
tv_list.DeletePictures()
```

See also

DeletePicture
AddPicture

DeleteRow

Description

Deletes a row from a DataWindow control, DataStore object, or child DataWindow.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**DeleteRow** (*row*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If any argument's value is NULL, DeleteRow returns NULL.

Usage

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

If the DataWindow 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*)

Argument	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. If any argument's value is NULL, DeleteSeries returns NULL.

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.

Examples

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

DeleteSmallPicture

Description

Deletes a small picture from a ListView control.

Applies to

ListView controls

Syntax

listviewname.**DeleteSmallPicture** (*index*)

Argument	Description
listviewname	The name of the ListView control from which you want to delete a small picture from the image list
index	The index number of the small picture you want to delete

Return value

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

Examples

This example deletes a small picture from a ListView control:

```
lv_list.DeleteSmallPicture(1)
```

See also

DeleteSmallPictures

DeleteSmallPictures

Description

Deletes all small pictures from a ListView control.

Applies to

ListView controls

Syntax

listviewname.**DeleteSmallPictures** ( )

Argument	Description
listviewname	The name of the ListView control from which you want to delete all small pictures

Return value

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

Examples

This example deletes all small pictures from a ListView control:

```
lv_list.DeleteSmallPictures()
```

See also

DeleteSmallPicture

DeleteStatePicture

Description

Deletes a state picture from a control.

Applies to

ListView and TreeView controls

Syntax

controlname.**DeleteStatePicture** (*index*)

Argument	Description
controlname	The name of the ListView or TreeView control from which you want to delete a picture from the state image list
index	The index number of the state picture you want to delete

Return value

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

Examples

This example deletes a state picture from a ListView control:

```
lv_list.DeleteStatePicture(1)
```

See also

DeleteStatePictures

DeleteStatePictures

Description

Deletes all state pictures from a control.

Applies to

ListView and TreeView controls

Syntax

controlname.**DeleteStatePictures** ( )

Argument	Description
controlname	The name of the ListView or TreeView control from which you want to delete all state pictures

Return value

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

Examples

This example deletes all state pictures from a ListView control:

```
lv_list.DeleteStatePictures()
```

See also

DeleteStatePicture

Describe

Description

Reports the values of properties of a DataWindow object and objects within the DataWindow object. Each column and graphic object in the DataWindow has a set of properties (listed in the *DataWindow Reference*). You specify one or more properties as a string, and Describe returns the values of the properties.

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

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**Describe** (*propertylist*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow for which you want information about the structure
propertylist	A string whose value is a blank-separated list of properties or Evaluate functions. For a list of valid properties, see the *DataWindow Reference*

Return value

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

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

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

If any argument's value is NULL, Describe returns NULL.

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 a property 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 property 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.

Property syntax The syntax for a property in the property list is:

objectname.property

FOR INFO For the types of objects in a DataWindow and their properties with examples, see the *DataWindow Reference*.

Properties whose value is a list When a property returns a list, the tab character (~t) separates the values in the list. For example, the Bands property reports all the bands in use in the DataWindow as a list.

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

If the first character in a property'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 property value.

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

Quoted property values
Describe returns a property's value enclosed in quotes when the text would otherwise be ambiguous. For example, if the property's value includes a question mark, then the text is returned in quotes. A question mark without quotes means that the property 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 properties 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 *propertylist*.

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* below.)

Sample property 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 properties and the returned value. In the first example below, a sample command shows how you might specify these properties for Describe and what it reports.

Property	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

This example calls Describe with the list of properties 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 This statement finds out the column type of the current column, using the column name:

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

For comparison, this 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 DropDownDataWindow This example uses Describe's Evaluate function to find the display value in a DropDownDataWindow column called state_code. 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 the return code. Data and row are arguments of the ItemError event:

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

CHOOSE CASE s
    CASE "number"
       IF Trim(data) = "" THEN
          integer null_num
          SetNull(null_num)
          This.SetItem(row, &
             This.GetColumn(), null_num)
          RETURN 3
       END IF

CASE "date"
       IF Trim(data) = "" THEN
       date null_date
```

```
            SetNull(null_date)
            This.SetItem(row, &
            This.GetColumn(), null_date)
            RETURN 3
        END IF

        . . . // Additional cases for other data types

    END CHOOSE
```

See also

Create
Modify

DestroyModel

Description

Destroys the current performance analysis or trace tree model.

Applies to

Profiling and TraceTree objects

Syntax

instancename.**DestroyModel** ( )

Argument	Description
instancename	Instance name of the Profiling or TraceTree object

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- ModelNotExistsError!—The function failed because no model exists

Usage

When you are finished with the performance analysis or trace tree model you created using the BuildModel function, you must call DestroyModel to destroy the model as well as all the objects associated with that model. The memory allocated to a model will not be released until the object is destroyed.

Examples

This example destroys the performance analysis model previously created using the BuildModel function:

```
lpro_model.DestroyModel()
DESTROY lpro_model
```

See also

BuildModel

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, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

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

Argument	Description
listboxname	The name of the ListBox control 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). If any argument's value is NULL, DirList returns NULL.

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.)

Alternatives
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)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this if some files use a file extension .doc:

```
lb_emp.DirList("HD:Employee:*.doc", 1, st_path)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
lb_emp.DirList("/home/export/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, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

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

Argument	Description
listboxname	The name of the ListBox control 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 pathname 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). If any argument's value is NULL, DirSelect returns NULL.

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.

Examples

The following script for the SelectionChanged event for the ListBox lb_FileList calls 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

Menu objects

Syntax

menuname.**Disable** ( )

Argument	Description
menuname	The name of the menu selection you want to deactivate (disable)

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *menuname* is NULL, Disable returns NULL.

Equivalent syntax Setting the menu's Enabled property is the same as calling Disable.

menuname.Enabled = FALSE

This statement:

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

is equivalent to:

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

Examples

This statement disables the m_edit menu item 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** ( )

Argument	Description
oleobject	The name of an OLEObject variable that you want to disconnect from an OLE object. You cannot specify an OLEObject that is the Object property of an OLE 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 property of a control
-9 Other error

If *oleobject* is NULL, DisconnectObject returns NULL.

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 INFO For more information about OLE automation, see ConnectToObject.

Examples

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

```
integer result
OLEObject myoleobject

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

See also

ConnectToObject
ConnectToNewObject

DisconnectServer

Description

Disconnects a client application from a server application.

This function applies to distributed applications only.

Applies to

Connection objects

Syntax

connection.**DisconnectServer** ( )

Argument	Description
connection	The name of the Connection object used to establish the connection you want to delete

Return value

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

50 Distributed service error
52 Distributed communications error
53 Requested server not active
54 Server not accepting requests
55 Request terminated abnormally
56 Response to request incomplete
57 Not connected
62 Server busy

Usage

After disconnecting from the server application, the client application needs to destroy the Connection object.

DisconnectServer causes all remote objects and proxy objects created for the client connection to be destroyed.

Examples

In this example, the client application disconnects from the server application using the Connection object myconnect:

```
myconnect.DisconnectServer()
destroy myconnect
```

See also

ConnectToServer

DoScript

Description

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

Platform information
DoScript has an effect on Macintosh only.

Syntax

DoScript (*script, result*)

Argument	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. If any argument's value is NULL, DoScript returns NULL.

Usage

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

Examples

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*)

Argument	Description
stringorblob	A 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. *Stringorblob* can also be an Any variable containing a double or blob

Return value

Double. Returns the contents of *stringorblob* as a double. If *stringorblob* is not a valid PowerScript number or if it contains a non-numeric data type, Double returns 0. If *stringorblob* is NULL, Double returns NULL.

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, this 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 control or OLE DWObject.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE controls and OLE DWObjects (objects within a DataWindow object that is within a DataWindow control)

Syntax

objectref.**DoVerb** (*verb*)

Argument	Description
objectref	The name of the OLE control or the fully qualified name of a OLE DWObject within a DataWindow control for which you want to execute a verb. The fully qualified name for a DWObject has this syntax: *dwcontrol*.Object.*dwobjectname*
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 Container 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

If any argument's value is NULL, DoVerb returns NULL.

Examples

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

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

This example executes verb 7 for the object in the OLE DWObject ole_graph:

```
integer result
result = dw_1.Object.ole_graph.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*)

Argument	Description
control	The name of the control you want to drag or stop dragging
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 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 controls, returns the following values:

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

If any argument's value is NULL, Drag returns NULL.

Usage

To see the list of draggable controls, open the Browser. All the objects in the hierarchy below dragobject are draggable.

If you set the control's DragAuto property 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).

Examples

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 properties of the control, you can assign the DragObject reference to a variable of that control's data type (see the example).

Examples

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 property 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*)

Argument	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. If any argument's value is NULL, Draw returns NULL. The return value is usually not used.

Usage

Using the Draw function is faster and produces less flicker than successively changing the X property of a picture. This is because the Draw function draws directly on the window rather than recreating 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 a 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 where 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 many 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

EditLabel

Put a label in a ListView or TreeView control into edit mode.

To enable editing of a label in a	Use
ListView control	Syntax 1
TreeView control	Syntax 2

Syntax 1

For editing a label in a ListView

Description

Puts a label in a ListView into edit mode.

Applies to

ListView controls

Syntax

listviewname.**EditLabel** (*index*)

Argument	Description
listviewname	The ListView control in which you want to enable label editing
index	The index of the ListView item to be edited

Return value

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

Usage

If the EditLabels property for the ListView is TRUE, the user can edit the label of an item by selecting the item and then clicking on the label.

If the EditLabels property is set to FALSE, calling the EditLabel function is the only way to allow the user to edit the label.

Examples

This example allows the user to edit the label of the first selected item in the ListView control lv_1:

```
integer li_selected
li_selected = lv_1.SelectedIndex()
lv_1.EditLabel(li_selected)
```

See also

FindItem

Syntax 2

For editing a label in a TreeView

Description

Puts a label in a TreeView into edit mode.

Applies to

TreeView controls

Syntax

treeviewname.**EditLabel** (*itemhandle*)

Argument	Description
treeviewname	The TreeView control in which you want to enable label editing
itemhandle	The handle of the item to be edited

Return value

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

Usage

If the EditLabels property for the TreeView is TRUE, the user can edit the label of an item by selecting the item and then clicking again to edit the label.

If the EditLabels property is set to FALSE, calling the EditLabel function is the only way to allow the user to edit the label.

Examples

This example allows the user to edit the label of the current TreeView item:

```
long ll_tvi
ll_tvi = tv_list.FindItem(CurrentTreeItem!, 0)
tv_list.EditLabel(ll_tvi)
```

See also

FindItem

Enable

Description

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

Applies to

Menu objects

Syntax

menuname.**Enable** ( )

Argument	Description
menuname	The name of the menu selection you want to enable

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *menuname* is NULL, Enable returns NULL.

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 property to TRUE.

Equivalent syntax Setting the menu's Enabled property is the same as calling Enable.

menuname.Enabled = TRUE

This statement:

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

is equivalent to:

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

Examples

This statement enables the m_delete menu selection on the menu m_appl:

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

See also

Disable

EntryList

Description

Provides a list of the top-level entries included in a trace tree model.

Applies to

TraceTree objects

Syntax

instancename.**EntryList** (*list*)

Argument	Description
instancename	Instance name of the TraceTree object
list	An unbounded array variable of data type TraceTreeNode in which EntryList stores a TraceTreeNode object for each top-level entry in the trace tree model. This argument is passed by reference

Return value

ErrorReturn. Returns the following values:

- Success!—The function succeeded
- ModelNotExistsError!—The function failed because no model exists

Usage

You use the EntryList function to extract a list of the top-level entries or nodes included in a trace tree model. Each top-level entry listed is defined as a TraceTreeNode object and provides the type of activity represented by that node.

You must have previously created the trace tree model from a trace file using the BuildModel function.

Examples

This example gets the top-level entries or nodes in a trace tree model and then loops through the list extracting information about each node. The of_dumpnode function takes a TraceTreeNode object and a level as arguments and returns a string containing information about the node:

```
TraceTree ltct_model
TraceTreeNode ltctn_list[], ltctn_node
Long ll_index,ll_limit
String ls_line

ltct_model = CREATE TraceTree
ltct_model.BuildModel()
```

```
ltct_model.EntryList(ltctn_list)
ll_limit = UpperBound(ltctn_list)
FOR ll_index = 1 TO ll_limit
   ltctn_node = ltctn_list[ll_index]
   ls_line += of_dumpnode(ltctn_node,0)
NEXT
...
```

See also BuildModel

EventParmDouble

Description

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

Platform information
This and other functions for VBX user objects have no effect on Macintosh, UNIX, and Windows NT.

Applies to

VBX user objects

Syntax

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

Argument	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. If any argument's value is NULL, EventParmDouble returns NULL.

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

Examples

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

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

See also

EventParmString

EventParmString

Description

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

Platform information
This and other functions for VBX user objects have no effect on Macintosh, UNIX, and Windows NT.

Applies to

VBX user objects

Syntax

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

Argument	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. If any argument's value is NULL, EventParmString returns NULL.

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

Examples

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

EventParmDouble

ExecRemote

Asks a DDE server application to execute the specified command.

To send	Use
A single command to a DDE server application (a cold link)	Syntax 1
A command to a DDE server application after you have opened a channel (a warm link)	Syntax 2

Syntax 1

For sending single commands

Description

Sends a single command to a DDE server application, called a **cold** link.

Platform information

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

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

ExecRemote (*command, applname, topicname*)

Argument	Description
command	A string whose value is the command you want a DDE server application to execute. To determine the correct command format, see the documentation for the server 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 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

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

If any argument's value is NULL, ExecRemote returns 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 use a 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 one that will acknowledge the request. The is also true for the related functions GetRemote and SetRemote.

A *warm link* is associated with a DDE channel (see Syntax 2).

A DDE hot link, which enables automatic updating of data in the PowerBuilder client application, involves other functions.

FOR INFO For more information, see the StartHotLink function.

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")
```

See also

CloseChannel
GetRemote
OpenChannel
SetRemote

Syntax 2 For commands over an opened channel

Description

Sends a command to a DDE server application when you have already called OpenChannel and established a warm link with the server.

Platform information

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

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology

Syntax

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

Argument	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

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 use 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 (see Syntax 1).A cold link

A *warm link* is associated with a DDE channel. You establish a channel for the DDE conversation with OpenChannel before sending commands with this syntax 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.

FOR INFO For more information, see the StartHotLink function.

Examples

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*)

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

Return value

Double. Returns *e* raised to the power *n*. If *n* is NULL, Exp returns NULL.

Inverse of Exp
The inverse of the Exp function is the Log function.

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 the *DataWindow Reference*

ExpandAll

Description

Recursively expands a specified item.

Applies to

TreeView controls

Syntax

treeviewname.**ExpandAll** (*itemhandle*)

Argument	Description
treeviewname	The TreeView control in which you want to expand an item and all the subordinate items in its hierarchy
itemhandle	The handle of the item you want to expand

Return value

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

Usage

To expand all levels in a TreeViewItem, use the ExpandAll function for the RootTreeItem.

Examples

This example expands all levels of a TreeView control:

```
long ll_tvi
ll_tvi = tv_list.FindItem(RootTreeItem! , 0)
tv_list.ExpandAll(ll_tvi)
```

See also

ExpandItem
CollapseItem

ExpandItem

Description

Expands a specified item.

Applies to

TreeView controls

Syntax

treeviewname.**ExpandItem** (*itemhandle*)

Argument	Description
treeviewname	The TreeView control in which you want to expand an item
itemhandle	The handle of the item you want to expand

Return value

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

Usage

ExpandItem will only expand a single item. To expand a specified item including its children, use ExpandAll.

Examples

This example expands the current level of a TreeView:

```
long ll_tvi
ll_tvi = tv_list.FindItem(CurrentTreeItem! , 0)
tv_list.ExpandItem(ll_tvi)
```

See also

ExpandAll
CollapseItem

Fact

Description

Determines the factorial of a number.

Syntax

Fact (*n*)

Argument	Description
n	The number for which you want the factorial

Return value

Double. Returns the factorial of *n*. If *n* is NULL, Fact returns NULL.

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 the *DataWindow Reference*

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#*)

Argument	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. If *file#* is NULL, FileClose returns NULL.

Examples

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*)

Argument	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. If *filename* is NULL, FileDelete returns NULL.

Examples

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*)

Argument	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. If *filename* is NULL, FileExists returns NULL.

Usage

If *filename* is locked by another application, causing a sharing violation, FileExists also returns FALSE.

Examples

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*)

Argument	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, you must 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. If *filename* is NULL, FileLength returns NULL.

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 32,765 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!)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
LengthA = FileLength("HD:East:EMP.TXT")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
LengthA = FileLength("\export\home\east\emp.txt")
```

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. If the file doesn't exist, a new file is created.

Syntax

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

Argument	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, you must 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 FOR INFO For more information, 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, but everyone has write access ♦ LockWrite! — Only the user who opened the file can write to it, but 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!

Argument	Description
creator (optional)	On Macintosh, a 4-character string specifying the creator of the file. If you specify *creator*, you must also specify *filetype*. On Macintosh, the file's creator string is case-sensitive—PWRS is different from pwrs or Pwrs. If the named file is not found and FileOpen creates a new file, *creator* becomes the file's creator. If you don't specify *creator* and FileOpen creates a new Macintosh file, the file's creator is set to ttxt (for TeachText or SimpleText). If *creator* is not a 4-character string, an execution error occurs. Otherwise, *creator* is ignored if the named file exists. *Creator* has no significance on other platforms
filetype (optional)	On Macintosh, a 4-character string specify the file's type. The filetype is case-sensitive. If the named file is not found and FileOpen creates a new file, *filetype* becomes the file's type. If you don't specify *filetype* and FileOpen creates a new Macintosh file, the file's type is set to TEXT. If *filetype* is not a 4-character string, an execution error occurs. Otherwise, *filetype* is ignored if the named file exists. *Filetype* has no significance on other platforms

Return value

Integer. Returns the file number assigned to *filename* if it succeeds and -1 if an error occurs. If any argument's value is NULL, FileOpen returns NULL.

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, whichever 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")
```

This 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!)
```

On Macintosh On Macintosh, the filename in the preceding code might be Employee Data and look like this:

```
li_FileNum = FileOpen("HD:Dept:Employee Data", &
    StreamMode!, Write!, LockWrite!, Replace!)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
li_FileNum = FileOpen( &
    "/home/export/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*)

Argument	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. If any argument's value is NULL, FileRead returns NULL.

Usage

If the file is opened in Line mode, FileRead 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.

An end-of-file mark is a NULL character (ASCII value 0). Therefore, if the file being read contains null characters, FileRead will stop reading at the first null character, interpreting it as the end of the file.

Unicode version of PowerBuilder
If a file is opened in Line mode in the Unicode version of PowerBuilder, FileRead reads the file as a Unicode file if it is in Unicode format or as an ANSI file if it is in ANSI format. If a file is opened in Stream mode, FileRead reads the file as a binary file.

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
```

On Macintosh On Macintosh, the filename in the preceding code might be Employee Data and look like this:

```
ll_FLength = FileLength( &
   "HD:Human Resources:Employee Data")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
ll_FLength = FileLength( &
   "/export/home/hr/emp_data.txt")
```

This 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)
```

This example reads a file exceeding 32,765 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
FileRead
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*)

Argument	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	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. If any argument's value is NULL, FileSeek returns NULL.

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*)

Argument	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. If any argument's value is NULL, FileWrite returns NULL.

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 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.

Length limit

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.

Unicode version of PowerBuilder

If a file is opened in Line mode in the Unicode version of PowerBuilder, FileWrite writes out the file as a Unicode file if it was a Unicode file or as an ANSI file if it was an ANSI file. A new file is written out as a Unicode file. If a file is opened in Stream mode, FileWrite writes out the file as a binary file.

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")
```

On Macintosh On Macintosh, the filename in the preceding code might be Employee Data and look like this:

```
ll_FLength = FileLength( &
   "HD:Human Resources:Employee Data")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
ll_FLength = FileLength( &
   "/export/home/hr/emp_data.txt")
```

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!, Shared!, 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*)

Argument	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. If any argument's value is NULL, Fill returns NULL.

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 the *DataWindow Reference*

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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**Filter** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow that you want to filter

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *dwcontrol* is NULL, Filter returns NULL. 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.

When the Retrieve As Needed option is set, the Filter function cancels its effect. Filter causes all rows to be retrieved and then it applies the filter.

Filter has no effect on the DataWindows in a composite report.

Filtering and groups
When you filter a DataWindow with groups, you may need to call GroupCalc after you call Filter.

FOR INFO For information on removing the filter or letting the user specify a filter expression, see SetFilter.

Examples

This statement displays rows in dw_Employee based on its current filter criteria:

```
dw_Employee.SetRedraw(false)
dw_Employee.Filter()
dw_Employee.SetRedraw(true)
```

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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**FilteredCount** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow for which you want the number of rows that are filtered out of the display

Return value

Long. Returns the number of rows in *dwcontrol* 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. If *dwcontrol* is NULL, FilteredCount returns NULL.

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.SetRedraw(false)
dw_Employee.Filter()
dw_Employee.SetRedraw(true)

// Did any rows get filtered out
IF dw_Employee.FilteredCount() > 0 THEN
. . . // Process rows not in area code 617
END IF
```

These 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.SetRedraw(false)
dw_Employee.Filter()
dw_Employee.SetRedraw(true)

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

Finds data in a DataWindow control or DataStore, or text in a RichTextEdit control or RichTextEdit DataWindow or DataStore.

To find	Use
Data in a row in a DataWindow or DataStore	Syntax 1
Text in a RichTextEdit control or RichTextEdit DataWindow or DataStore	Syntax 2

Syntax 1

For DataWindows and DataStores

Description

Finds the next row in a DataWindow or DataStore in which data meets a specified condition.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**Find** (*expression*, *start*, *end*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. *Start* can be greater than the number of rows
end	A long identifying the row location at which to end the search. *End* can be greater than the number of rows. To search backward, make *end* less than *start*

Return value

Long. Returns the number of the first row that meets the search criteria within the search range. Returns 0 if no rows are found and a negative number if an error occurs. If any argument's value is NULL, Find returns NULL.

Usage

The search is case-sensitive. When you compare text to a value in a column, the case must match.

If you use Find in a loop that searches through all rows, you may end up with an endless loop if the last row satisfies the search criteria. When the *start* value becomes greater than *end*, the search reverses direction and Find would always succeed, resulting in an endless loop.

To solve this problem, you could make the *end* value 1 greater than the number of rows (see the examples). Another approach, shown below, would be to test within the loop whether the current row is greater than the row count and, if so, exit:

```
long ll_find = 1, ll_end
ll_end = dw_main.RowCount()
ll_find = dw_main.Find(searchstr, ll_find, ll_end)
DO WHILE ll_find > 0
    ... // Collect found row
    ll_find++
    // Prevent endless loop
    IF ll_find > ll_end THEN EXIT
    ll_find = dw_main.Find(searchstr, ll_find, ll_end)
LOOP
```

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:

```
long ll_found
ll_found = 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:

```
long ll_found
ll_found = 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 a Find Next command button 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 num. 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
```

This example searches all the rows in dw_main and builds a list of the names that include a lowercase a. Note that the end value of the search is one greater than the row count, avoiding an infinite loop if the name in the last row satisfies the search:

```
long ll_find, ll_end
string ll_list

// The end value is one greater than the row count
ll_end = dw_main.RowCount() + 1
ll_find = 1

ll_find = dw_main.Find("Pos(last_name,'a') > 0", &
    ll_find, ll_end)
DO WHILE ll_find > 0
    //collect names
    ll_list = ll_list + '~r' &
        + dw_main.GetItemString(ll_find, 'last_name')

    // Search again
    ll_find++
    ll_find = dw_main.Find("Pos(last_name,'a') > 0",&
        ll_find, ll_end )
LOOP
```

See also

FindGroupChange
FindRequired

Syntax 2

For text in RichTextEdit format

Description

Finds the specified text in the control and highlights the text if found. You can specify search direction and whether to match whole words and case.

Applies to

RichTextEdit controls and DataWindow controls (or DataStore objects) whose content has the RichTextEdit presentation style

Syntax

controlname.**Find** (*searchtext, forward, insensitive, wholeword, cursor*)

Argument	Description
controlname	The name of the RichTextEdit, DataWindow control, or DataStore whose contents you want to search
searchtext	A string whose value is the text you want to find.
forward	A boolean value indicating the direction you want to search. Values are: ♦ TRUE — The search proceeds forward from the cursor position or, if *cursor* is false, from the start of the document ♦ FALSE — The search proceeds backward from the cursor position or, if *cursor* is false, from the end of the document
insensitive	A boolean value indicating the search string and the found text must match case. Values are: ♦ TRUE — The search is not sensitive to case ♦ FALSE — The search is case-sensitive
wholeword	A boolean value indicating that the found text must be a whole word. Values are: ♦ TRUE — The found text must be a whole word ♦ FALSE — The found text can be a partial word
cursor	A boolean value indicating where the search begins. Values are: ♦ TRUE — The search begins at the cursor position ♦ FALSE — The search begins at the start of the document if *forward* is TRUE or at the end if *forward* is FALSE

Return value

Integer. Returns the number of characters found. Find returns 0 if no matching text is found, and returns -1 if the DataWindow's presentation style is not RichTextEdit or an error occurs.

Examples

This example searches the RichTextEdit rte_1 for text the user specifies in the SingleLineEdit sle_search. The search proceeds forward from the cursor position. The search is case-insensitive and not limited to whole words:

```
integer li_charsfound
li_charsfound = rte_1.Find(sle_search.Text, &
   TRUE, TRUE, FALSE, TRUE)
```

See also

FindNext

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 and DataStore objects

Syntax

controlname.**FindCategory** ({ *graphcontrol*, } *categoryvalue*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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 or DataStore, in *graphcontrol*. If an error occurs, FindCategory returns -1. If any argument's value is NULL, FindCategory returns NULL.

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

FindClassDefinition

Description

Searches for an object in one or more PowerBuilder libraries (PBLs) and provides information about its class definition.

Syntax

FindClassDefinition (*classname* {, *librarylist* })

Argument	Description
classname	The name of an object (also called a class or class definition) for which you want information
librarylist (optional)	An array of strings whose values are the fully qualified pathnames of PBLs. If you omit *librarylist*, FindClassDefinition searches the library list associated with the running application

Return value

ClassDefinition. Returns an object reference with information about the definition of *classname*. If any arguments are NULL, FindClassDefinition returns NULL.

Usage

There are two ways to get a ClassDefinition object containing class definition information:

- For an instantiated object in your application, use its ClassDefinition property
- For an object stored in a PBL, call FindClassDefinition

Examples

This example searches the libraries for the running application to find the class definition for w_genapp_frame:

```
ClassDefinition cd_windef
cd_windef = FindClassDefinition("w_genapp_frame")
```

This example searches the libraries in the array ls_libraries to find the class definition for w_genapp_frame:

```
ClassDefinition cd_windef
string ls_libraries[ ]

ls_libraries[1] = "c:\pwrs\bizapp\windows.pbl"
ls_libraries[2] = "c:\pwrs\framewk\windows.pbl"
ls_libraries[3] = "c:\pwrs\framewk\ancestor.pbl"

cd_windef = FindClassDefinition(
   "w_genapp_frame", ls_libraries)
```

See also

FindFunctionDefinition
FindMatchingFunction
FindTypeDefinition

FindFunctionDefinition

Description

Searches for a global function in one or more PowerBuilder libraries (PBLs) and provides information about the script definition.

Syntax

FindFunctionDefinition (*functionname* {, *librarylist* })

Argument	Description
functionname	The name of a global function for which you want information
librarylist (optional)	An array of strings whose values are the fully qualified pathnames of PBLs. If you omit *librarylist*, FindFunctionDefinition searches the library list associated with the running application

Return value

ScriptDefinition. Returns an object reference with information about the script of *functionname*. If any arguments are NULL, FindFunctionDefinition returns NULL.

Usage

You can call FindClassDefinition to get a class definition for a global function. However, the ScriptDefinition object provides information tailored for functions.

Examples

This example searches the libraries for the running application to find the function definition for f_myfunction:

```
ScriptDefinition sd_myfunc
sd_myfunc = FindFunctionDefinition("f_myfunction")
```

This example searches the libraries in the array ls_libraries to find the class definition for w_genapp_frame:

```
ScriptDefinition sd_myfunc
string ls_libraries[ ]

ls_libraries[1] = "c:\pwrs\bizapp\windows.pbl"
ls_libraries[2] = "c:\pwrs\framewk\windows.pbl"
ls_libraries[3] = "c:\pwrs\framewk\ancestor.pbl"

sd_myfunc = FindFunctionDefinition( &
   "f_myfunction", ls_libraries)
```

See also

FindClassDefinition
FindMatchingFunction
FindTypeDefinition

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 DataStore objects

Syntax

dwcontrol.**FindGroupChange** (*row, level*)

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore 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, meaning that it begins a new group. Returns 0 if the value in the group column did not change and a negative number if an error occurs. If any argument's value is NULL, FindGroupChange returns NULL.

The return value observes these rules based on the value of *row*. If the starting row is:

- The first row in a group, then FindGroupChange returns the starting row number
- A row within a group, other than the last group, then FindGroupChange returns the row number of the first row of the next group
- A row in the last group, other than the first row of the last group, then FindGroupChange returns 0

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

Finds the next item in a list.

To find the next item	Use
In a ListBox, DropDownListBox, PictureListBox, or DropDownPictureListBox	Syntax 1
In a ListView control based upon its label	Syntax 2
By relative position in a ListView control	Syntax 3
By relative position in a TreeView control	Syntax 4

Syntax 1

For ListBox and DropDownListBox controls

Description

Finds the next item in a ListBox that begins with the specified search text.

Applies to

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

listboxname.**FindItem** (*text*, *index*)

Argument	Description
listboxname	The name of the ListBox control 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 just before the first item to be searched. To search the whole list, specify 0

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. If any argument's value is NULL, FindItem returns NULL.

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.

Examples

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", 1)
```

See also

AddItem
DeleteItem
InsertItem
SelectItem

Syntax 2

For ListView controls

Description

Searches for the next item whose label matches the specified search text.

Applies to

ListView controls

Syntax

listviewname.**FindItem** (*startindex, label, partial, wrap*)

Argument	Description
listviewname	The ListView control for which you want to search for items
startindex	The index number from which you want your search to begin
label	The string that is the target of the search
partial	If set to TRUE, the search looks for a partial label match
wrap	If set to TRUE, the search returns to the first index item after it has finished

Return value

Integer. Returns the index of the item found if it succeeds and -1 if an error occurs.

Usage

The search starts from *startindex* + 1 by default. To search from the beginning, specify 0.

If *partial* is set to TRUE, the search string matches any label that begins with the specified text. If *partial* is set to FALSE, the search string must match the entire label.

If *wrap* is set to TRUE, the search wraps around to the first index item after searching to the end. If *wrap* is set to FALSE, the search stops at the last index item in the ListView.

FindItem does not select the item it finds. You must use the item's selected property in conjunction with FindItem to select the resulting match.

Examples

This example takes the value from a SingleLineEdit control and passes it to FindItem:

```
listviewitem l_lvi
integer li_index
string ls_label

ls_label = sle_find.Text

IF ls_label = "" THEN
   MessageBox("Error" , &
      "Enter the name of a list item")
   sle_find.SetFocus()
ELSE
   li_index = lv_list.FindItem(0,ls_label, TRUE,TRUE)
END IF

IF li_index =  1 THEN
   MessageBox("Error", "Item not found.")
ELSE
   lv_list.GetItem (li_index, l_lvi )
   l_lvi.HasFocus = TRUE
   l_lvi.Selected = TRUE
   lv_list.SetItem(li_index,l_lvi)
END IF
```

See also

AddItem
DeleteItem
InsertItem
SelectItem

Syntax 3

For ListView controls

Description

Search for the next item relative to a specific location in the ListView control.

Applies to

ListView controls

Syntax

listviewname.**FindItem** (*startindex, direction, focused, selected, cuthighlighted, drophighlighted*)

Argument	Description
listviewname	The ListView control for which you want to search for items
startindex	The index number from which you want your search to begin
direction	The direction in which to search. Values are: DirectionAll! DirectionUp! DirectionDown! DirectionLeft! DirectionRight!
focused	If set to TRUE, the search looks for the next ListView item that has focus
selected	If set to TRUE, the search looks for the next ListView item that is selected
cuthighlighted	If set to TRUE, the search looks for the next ListView item that is the target of a cut operation
drophighlighted	If set to TRUE, the search looks for next ListView item that is the target of a drag and drop operation

Return value

Integer. Returns the index of the item found if it succeeds and -1 if an error occurs.

Usage

The search starts from *startindex* + 1 by default. If you want to search from the beginning, specify 0.

FindItem does not select the item it finds. You must use the item's selected property in conjunction with FindItem to select the resulting match.

If *focused*, *selected*, *cuthighlighted*, and *drophighlighted* are set to FALSE, the search finds the next item in the ListView control.

Examples

This example uses FindItem to search from the selected ListView item:

```
listviewitem l_lvi
integer li_index li_startindex
```

```
li_startindex = lv_list.SelectedIndex()
li_index = lv_list.FindItem(li_startindex, &
    DirectionDown!, FALSE, FALSE ,FALSE, FALSE)

IF li_index = -1 THEN
    MessageBox("Error", "Item not found.")
ELSE
    lv_list.GetItem (li_index, l_lvi)
    l_lvi.HasFocus = TRUE
    l_lvi.Selected = TRUE
    lv_list.SetItem(li_index,l_lvi)
END IF
```

See also

AddItem
DeleteItem
InsertItem
SelectItem

Syntax 4

For TreeView controls

Description

Find an item based on its position in a TreeView control.

Applies to

TreeView controls

Syntax

treeviewname.**FindItem** (*navigationcode, itemhandle*)

Argument	Description
treeviewname	The name of the TreeView control in which you want to find a specified item

Argument	Description
navigationcode	A value of the TreeNavigation enumerated data type specifying the relationship between *itemhandle* and the item you want to find. Valid values are: ♦ RootTreeItem! — Finds the first item at level 1. Returns -1 if no items have been inserted into the control ♦ NextTreeItem! — Finds the sibling after *itemhandle*. A sibling is an item at the same level with the same parent. Returns -1 if there are no more siblings ♦ PreviousTreeItem! — Finds the sibling before *itemhandle*. Returns -1 if there are no more siblings ♦ ParentTreeItem! — Finds the parent of *itemhandle*. Returns -1 if the item is at level 1 ♦ ChildTreeItem! — Finds the first child of *itemhandle*. If the item is collapsed, ChildtreeItem! causes the node to expand. Returns -1 if the item has no children or if the item is not populated yet ♦ FirstVisibleTreeItem! — Finds the first item visible in the control, regardless of level. The position of the scrollbar determines the first visible item ♦ NextVisibleTreeItem! — Finds the next expanded item after *itemhandle*, regardless of level. NextVisible and PreviousVisible allows you to walk through all the children and branches of an expanded node. When the next item is beyond the visible area of the control, NextVisible causes the control to scroll to reach the next item. Returns -1 if the item is the last expanded item in the control ♦ PreviousVisibleTreeItem! — Finds the next expanded item before *itemhandle*, regardless of level. When the next item is beyond the visible area of the control, PreviousVisible causes the control to scroll to reach the next item. Returns -1 if the item is the first root item ♦ CurrentTreeItem! — Finds the selected item. Returns -1 if the control never had focus and nothing has been selected ♦ DropHighlightTreeItem! — Finds the item whose DropHighlighted property was most recently set. Returns -1 if the property was never set or if it has been set back to FALSE because of other activity in the control
itemhandle	The handle for an item that is related via *navigationcode* to the item for which you are searching. If *navigationcode* is RootTreeItem!, FirstVisibleTreeItem!, CurrentTreeItem!, or DropHighlightTreeItem!, set *itemhandle* to 0

Return value

Long. Returns the item handle if it succeeds and -1 if an error occurs.

Usage

FindItem does not select the item it finds. You must use the item's selected property in conjunction with FindItem to select the result of the FindItem search.

FindItem never finds a collapsed item, except when looking for ChildTreeItem!, which causes an item to expand.

Examples

This example finds the currently selected item in a TreeView control:

```
long ll_tvi
ll_tvi = tv_list.FindItem(CurrentTreeItem!, 0)
```

This example finds the first item on the first level of a TreeView control:

```
long ll_tvi
ll_tvi = tv_list.FindItem(RootTreeItem!, 0)
```

See also

DeleteItem
GetItem
InsertItem
SelectItem

FindMatchingFunction

Description

Finds out what function in a class matches a specified signature. The signature is a combination of a script name and an argument list.

Applies to

ClassDefinition objects

Syntax

classdefobject.**FindMatchingFunction** (*scriptname, argumentlist*)

Argument	Description
classdefobject	The name of the ClassDefinition object describing the class in which you want to find a function
scriptname	A string whose value is the name of the function
argumentlist	An unbounded array of strings whose values are the data types of the function arguments. If the variable is passed by reference, the string must include "ref" before the data type. If the variable is an array, you must include array brackets after the data type The format is: { ref } *datatype* { [ ] }

Return value

ScriptDefinition. Returns an object instance with information about the matching function. If no matching function is found, FindMatchingFunction returns NULL. If any argument is NULL, it also returns NULL.

Usage

In searching for the function, PowerBuilder examines the collapsed inheritance hierarchy. The found function may be defined in the current object or in any of its ancestors.

Arguments passed by reference To find a function with an argument that is passed by reference, you must specify the "ref" keyword. If you have a VariableDefinition object for a function argument, check the CallingConvention argument to determine if the argument is passed by reference.

In documentation for PowerBuilder functions, arguments passed by reference are described as a variable, rather than simply a value. The PowerBuilder Browser does not report which arguments are passed by reference.

Examples

This example gets the ScriptDefinition object that matches the PowerBuilder window object function OpenUserObjectWithParm and looks for the version with four arguments. If it finds a match, the example calls the function uf_scriptinfo, which creates a report about the script:

```
string ls_args[]
ScriptDefinition sd

ls_args[1] = "ref dragobject"
ls_args[2] = "double"
ls_args[3] = "integer"
ls_args[4] = "integer"

sd = c_obj.FindMatchingFunction( &
    "OpenUserObjectWithParm", ls_args)

IF NOT IsValid(sd) THEN
    mle_1.Text = "No matching script"
ELSE
    mle_1.Text = uf_scriptinfo(sd)
END IF
```

The uf_scriptinfo function gets information about the function that matched the signature and builds a string. Scriptobj is the ScriptDefinition object passed to the function:

```
string s, lineend
integer li

lineend = "~r~n"

// Script name
s = s + scriptobj.Name + lineend
// Data type of the return value
s = s + scriptobj.ReturnType.DataTypeOf + lineend

// List argument names
s = s + "Arguments:" + lineend
FOR li = 1 to UpperBound(scriptobj.ArgumentList)
    s = s + scriptobj.ArgumentList[li].Name + lineend
NEXT
```

```
// List local variables
s = s + "Local variables:" + lineend
FOR li = 1 to UpperBound(scriptobj.LocalVariableList)
   s = s + scriptobj.LocalVariableList[li].Name &
      + lineend
NEXT

RETURN s
```

See also

FindClassDefinition
FindFunctionDefinition
FindTypeDefinition

FindNext

Description

Finds the next occurrence of text in the control and highlights it, using criteria set up in a previous call of the Find function.

Applies to

RichTextEdit controls and DataWindow controls whose content has the RichTextEdit presentation style

Syntax

controlname.**FindNext** ( )

Argument	Description
controlname	The name of the RichTextEdit or DataWindow control whose contents you want to search

Return value

Integer. Returns the number of characters found. Find returns 0 if no matching text is found and -1 if the DataWindow's presentation style is not RichTextEdit or an error occurs.

Examples

This example searches the RichTextEdit rte_1 for text the user specifies in the SingleLineEdit sle_search. The search proceeds forward from the cursor position, is case-insensitive, and is not limited to whole words:

```
integer li_charsfound
li_charsfound = rte_1.Find(sle_search.Text, &
   TRUE, TRUE, FALSE, TRUE)
```

A second button labeled Find Next would have a script like this:

```
rte_1.FindNext()
```

See also

Find

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 and DataStore objects

Syntax

dwcontrol.**FindRequired** (*dwbuffer, row, colnbr, colname, updateonly*)

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore in which you want to 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! ♦ Filter!
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 that contains 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 that contains 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 that 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. If any argument's value is NULL, FindRequired returns NULL.

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 property 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. If *updateonly* is FALSE, FindRequired will report empty required columns in new rows. If *updateonly* is TRUE, FindRequired does not check new rows because new, empty rows are not updated in the database.

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 property 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 = 0
string colname

mle_required.Text = ""
DO WHILE row <> 0
   colnbr++// Continue searching at next column
   // 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

   // When FindRequired returns 0 (meaning
   // no more rows found), drop out of 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 in the function declaration like this:

```
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) < 1 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 and DataStore objects

Syntax

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

Argument	Description
controlname	The name of the graph containing the series for which you want the number, or the name of the DataWindow control or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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 or DataStore, in *graphcontrol*. If an error occurs, FindSeries returns -1. If any argument's value is NULL, FindSeries returns NULL.

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

FindTypeDefinition

Description

Searches for a type in one or more PowerBuilder libraries (PBLs) and provides information about its type definition. You can also get type definitions for system types.

Syntax

FindTypeDefinition (*typename* {, *librarylist* })

Argument	Description
typename	The name of a simple data type, enumerated data type, or class for which you want information. To find a type definition for a nested type, use this form: *libraryEntryName`typename*
librarylist (optional)	An array of strings whose values are the fully qualified pathnames of PBLs. If you omit *librarylist*, FindTypeDefinition searches the library list associated with the running application PowerBuilder also searches its own libraries for built-in definitions, such as enumerated data types and system classes

Return value

TypeDefinition. Returns an object reference with information about the definition of *typename*. If any arguments are NULL, FindTypeDefinition returns NULL.

Usage

The returned TypeDefinition object will actually be a ClassDefinition, SimpleTypeDefinition, or EnumerationDefinition object. You can test the Category property to find out which one it is.

If you want to get information for a class, call FindClassDefinition instead. The arguments are the same and you are saved the step of checking that the returned object is a ClassDefinition object.

If you want to get information for a global function, call FindFunctionDefinition.

Examples

This example gets a TypeDefinition object for the grGraphType enumerated data type. It checks the category of the type definition and, since it's an enumeration, assigns it to an EnumerationDefinition object type and saves the name in a string:

```
TypeDefinition td_graphtype
EnumerationDefinition ed_graphtype
string enumname
```

```
td_graphtype = FindTypeDefinition("grgraphtype")
IF td_graphtype.Category = EnumeratedType! THEN
   ed_graphtype = td_graphtype
   enumname = ed_graphtype.Enumeration[1].Name
END IF
```

This example is a function that takes a definition name as an argument. The argument is typename. It finds the named TypeDefinition object, checks its category, and assigns it to the appropriate definition object:

```
TypeDefinition td_def
SimpleTypeDefinition std_def
EnumerationDefinition ed_def
ClassDefinition cd_def

td_def = FindTypeDefinition(typename)
CHOOSE CASE td_def.Category
CASE SimpleType!
   std_def = td_def
CASE EnumeratedType!
   ed_def = td_def
CASE ClassOrStructureType!
   cd_def = td_def
END CHOOSE
```

This example searches the libraries in the array ls_libraries to find the class definition for w_genapp_frame:

```
TypeDefinition td_windef
string ls_libraries[ ]

ls_libraries[1] = "c:\pwrs\bizapp\windows.pbl"
ls_libraries[2] = "c:\pwrs\framewk\windows.pbl"
ls_libraries[3] = "c:\pwrs\framewk\ancestor.pbl"

td_windef = FindTypeDefinition(
   "w_genapp_frame", ls_libraries)
```

See also

FindClassDefinition
FindFunctionDefinition
FindMatchingFunction

GarbageCollect

Description	Forces immediate garbage collection.
Syntax	**GarbageCollect** ( )
Return value	None
Usage	Forces garbage collection to occur immediately. PowerBuilder makes a pass to identify unused objects, including those with circular references, then deletes unused objects and classes.
Examples	This statement initiates garbage collection: `GarbageCollect ( )`
See also	GarbageCollectGetTimeLimit GarbageCollectSetTimeLimit

GarbageCollectGetTimeLimit

Description

Gets the current minimum interval for garbage collection.

Syntax

GarbageCollectGetTimeLimit ( )

Return value

Long. Returns the current minimum garbage collection interval.

Usage

Reads the current minimum period between garbage collection passes.

Examples

This statement returns the interval between garbage collection passes in the variable CollectTime:

```
long CollectTime

CollectTime = GarbageCollectGetTimeLimit()
```

See also

GarbageCollect
GarbageCollectSetTimeLimit

GarbageCollectSetTimeLimit

Description

Sets the minimum interval between garbage collection passes.

Syntax

GarbageCollectSetTimeLimit (*newtimeinmilliseconds*)

Argument	Description
newtimeinmilliseconds	A long (in milliseconds) that you want to set as the minimum period between garbage collection cycles If NULL, the existing interval is not changed

Return value

Long. Returns the interval that existed before this function was called. If *newTime* is NULL, then NULL is returned and the current interval is not changed.

Usage

Specifies the minimum interval between garbage collection passes: garbage collection passes will not happen before this interval has expired.

Garbage collection can effectively be disabled by setting the minimum limit to a very large number. If garbage collection is disabled, unused classes will not be flushed out of the class cache.

Examples

This example sets the interval between garbage collection passes:

```
long NewTime
Time = 1000 /* 1 second */

NewTime = GarbageCollectSetTimeLimit(Time)
```

See also

GarbageCollect
GarbageCollectGetTimeLimit

GenerateHTMLForm

Description

Creates an HTML Form element containing columns for one or more rows in a DataWindow control or DataStore. This function also returns an HTML Style element containing style sheet information.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GenerateHTMLForm** (*syntax*, *style*, *action* { , *startrow*, *endrow*, *startcolumn*, *endcolumn* {, *buffer* } })

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore containing the DataWindow object whose contents are converted into HTML Form syntax
syntax	String into which the function places HTML Form syntax. This value is passed by reference
style	String into which the function places HTML style sheet. This value is passed by reference
action	String specifying the ACTION attribute for the Form element
startrow (optional)	Integer specifying the first row to be converted to HTML Form syntax. The default is 1
endrow (optional)	Integer specifying the last row to be converted to HTML Form syntax. The default is the last row in the DataWindow
startcolumn (optional)	Integer or string specifying the first column to be converted to HTML Form syntax. The default is the first column
endcolumn (optional)	Integer or string specifying the last column to be converted to HTML Form syntax. The default is the last column
buffer (optional)	dwBuffer enumerated data type specifying the buffer from which the function converts rows: ♦ Primary! (default) ♦ Delete! ♦ Filter!

Return value

Integer. Returns the number of bytes in *syntax* if the function succeeds and -1 if an error occurs.

Usage

Call this function to create HTML Form syntax from the contents of a DataWindow control or DataStore. After calling this function, you must define a complete HTML page by combining the Form syntax and style sheet with other appropriate HTML elements.

Examples

This Web.PB example calls the SaveAsHTMLPage function and then creates a complete HTML page and returns it to the Web browser:

```
String ls_syntax, ls_style, ls_action
String ls_html
Integer li_return

ls_action = &
   "/cgi-bin/pbcgi60.exe/myapp/uo_webtest/f_emplist"
li_return = ds_1.GenerateHTMLForm &
   (ls_syntax, ls_style, ls_action)
IF li_return = -1 THEN
   ls_html = "Unable to create HTML form."
ELSE
   ls_html = "<HTML>"
   ls_html += ls_style
   ls_html += "<BODY>"
   ls_html += "<H1>Employee Information</H1>"
   ls_html += ls_syntax
   ls_html += "</BODY></HTML>"
END IF
RETURN ls_html
```

See also

SaveAs

GetActiveSheet

Description

Returns the currently active sheet in an MDI frame window.

Applies to

MDI frame windows

Syntax

mdiframewindow.**GetActiveSheet** ( )

Argument	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. If *mdiframewindow* is NULL, GetActiveSheet returns NULL.

Usage

Use the IsValid function to determine whether GetActiveSheet has returned a valid window value.

Examples

These statements determine the active sheet in the MDI frame window w_frame and change the text of the menu selection m_close on the menu m_file on the menu bar m_main. If no sheet is active, the text 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 menu to read
   // Close plus the title of the active sheet
   mtext = "Close " + activesheet.Title
   m_main.m_file.m_close.Text = mtext

ELSE
   // No sheet is active, menu says Close Window
   m_main.m_file.m_close.Text = "Close Window"
END IF
```

See also

IsValid

GetAlignment

Description

Obtains the alignment of the paragraph containing the insertion point in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**GetAlignment** ( )

Argument	Description
rtename	The name of the RichTextEdit control in which you want to find out the alignment of the paragraph containing the insertion point

Return value

Alignment. A value of the Alignment enumerated data type indicating the alignment of the paragraph containing the insertion point.

Usage

When several paragraphs are selected, the insertion point is at the beginning or end of the selection, depending on how the user made the selection. The value reported depends on the location of the insertion point.

Examples

This examples saves the alignment setting of the paragraph that contains the insertion point:

```
alignment l_align
l_align = rte_1.GetAlignment()
```

See also

GetSpacing
GetTextStyle
SetAlignment
SetSpacing
SetTextStyle

GetApplication

Description

Gets the handle of the current Application object so you can get and set properties 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 properties.

Examples

To change whether Toolbar Tips are displayed, you can get the handle of the application object and set the ToolbarTips property:

```
application app
app = GetApplication()
app.ToolbarTips = FALSE
```

The previous example could be coded more simply as follows:

```
GetApplication().ToolbarTips = FALSE
```

GetArgElement

Description

Returns the value in the specified argument.

Applies to

Window ActiveX controls

Syntax

activexcontrol.**GetArgElement** (*index*)

Argument	Description
activexcontrol	Identifier for the instance of the PowerBuilder window ActiveX control. When used in HTML, the ActiveX control is the NAME attribute of the OBJECT element. When used in other environments, references the control that contains the PowerBuilder window ActiveX
index	Integer specify the argument to return

Return value

Any. Returns the specified argument.

Usage

Call this function after calling InvokePBFunction or TriggerPBEvent to access the updated value in an argument passed by reference.

JavaScript scripts must use this function to access arguments passed by reference. VBScript scripts can use this function if they established the argument list via calls to SetArgElement.

Examples

This JavaScript example calls the GetArgElement function:

```
...
   theArg = f.textToPB.value;
   PBRX1.SetArgElement(1, theArg);
   theFunc = "of_argref";
   retcd = PBRX1.InvokePBFunction(theFunc, numargs);
   rc = parseInt(PBRX1.GetLastReturn());
   IF (rc != 1) {
      alert("Error. Empty string.");
   }
   backByRef = PBRX1.GetArgElement(1);
...
```

See also

GetLastReturn
InvokePBFunction
SetArgElement
TriggerPBEvent

GetAutomationNativePointer

Description

Gets a pointer to the OLE object associated with the OLEObject variable. The pointer lets you call OLE functions in an external DLL for the object.

Applies to

OLEObject

Syntax

oleobject.**GetAutomationNativePointer** (*pointer*)

Argument	Description
oleobject	The name of an OLEObject variable containing the object for which you want the native pointer
pointer	A UnsignedLong variable in which you want to store the pointer. If GetAutomationNativePointer cannot get a valid pointer, *pointer* is set to 0

Return value

Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage

Using the pointer in your own DLL calls *Pointer* is a pointer to OLE's IUnknown interface. You can use it with QueryInterface() to get other interface pointers.

When you call GetAutomationNativePointer, PowerBuilder calls OLE's AddRef() function, which locks the pointer. You can release the pointer in your DLL function or in a PowerBuilder script with the ReleaseAutomationNativePointer function.

Examples

This example creates an OLEObject object, connects to an automation server, and gets a pointer for making external function calls. After processing, the pointer is released:

```
OLEObject oleobj_report
UnsignedLong lul_oleptr
integer li_rtn

oleobj_report = CREATE OLEObject
oleobj_report.ConnectToObject("report.doc")

li_rtn = &
oleobj_report.GetAutomationNativePointer(lul_oleptr)
IF li_rtn = 0 THEN
    ... // Call external functions for automation
    oleobj_report.&
       ReleaseAutomationNativePointer(lul_oleptr)
END IF
```

See also

GetNativePointer
ReleaseAutomationNativePointer
ReleaseNativePointer

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

dwcontrol.**GetBandAtPointer** ( )

Argument	Description
dwcontrol	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 in Usage). Returns the empty string ("") if an error occurs. If *dwcontrol* is NULL, GetBandAtPointer returns NULL.

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
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.

Examples

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 or DataStore object.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetBorderStyle** (*column*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol* as a value of the Border enumerated data type. Possible values are:

Box!
NoBorder!
ShadowBox!
Underline!

Returns NULL if it fails or if any argument's value is NULL.

Examples

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

GetChanges

Description

Retrieves changes made to a DataWindow or DataStore into a blob.

This function is used primarily in distributed applications.

Applies to

DataWindow controls and DataStore objects

Syntax

dwcontrol.**GetChanges** (*changeblob* {, *cookie* })

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore for which you want to get changes
changeblob	The blob into which the returned DataWindow changes will be placed
cookie (optional)	A read-only blob created by GetStateStatus that is compared with the changeblob to determine the likely success of a subsequent call to SetChanges

Return value

Long. Returns the number of rows in the DataWindow change blob if it succeeds and one of the following values if it fails:

- -1 An internal error occurred
- -2 There is a conflict between the state of the DataWindow change blob and the state of the DataWindow from which the cookie was created; an attempt to use this blob in a SetChanges call against the DataWindow will fail
- -3 There is a conflict between the state of the DataWindow change blob and the state of the DataWindow from which the cookie was created; but partial changes from the change blob can be applied

Usage

GetChanges is used in conjunction with SetChanges to synchronize two or more DataWindows or DataStores. GetChanges retrieves data buffers and status flags for changed rows in a DataWindow or DataStore and places this information in a blob. SetChanges then applies the contents of this blob to another DataWindow or DataStore.

Ordinarily, the change blob created by GetChanges includes only those rows that have a status of New!, NewModified!, or DataModified!. But if you call GetChanges just after a partially successful Update has occurred and the AutoCommit transaction property is set to True, the resulting change blob will include rows that were reset to NotModified! as a result of successful database updates.

In situations where a single DataStore on a server acts as the source for multiple target DataWindows (or DataStores) on different clients, you can use GetChanges in conjunction with GetStateStatus to determine the likely success of SetChanges. This allows you to avoid shipping a change blob across the wire when the SetChanges call will fail anyway (because changes in the blob conflict with changes made previously by another client).

To determine the likely success of SetChanges, you need to:

1 Call the GetStateStatus function on the DataStore on which you want to do a SetChanges. GetStateStatus checks the state of the DataStore and makes the state information available in a reference argument called a **cookie**. The cookie is generally much smaller than a DataWindow change blob.

2 Send the cookie back to the client.

3 Call the GetChanges function on the DataWindow from which you want to apply changes, passing the cookie retrieved from GetStateStatus as a parameter. The return value from GetChanges indicates whether there are currently any potential conflicts between the state of the DataWindow blob and the state of the DataStore.

If the return value from GetChanges indicates that there are potential conflicts, you can then be certain that a subsequent call to SetChanges will fail if the FailOnAnyConflict! argument is specified. However, if the return value from GetChanges indicates no conflicts, the call to SetChanges may *still* fail, because the state of the DataStore may have changed since you called GetStateStatus and GetChanges. For example, if another client session has called SetChanges or some other processing has been executed that altered the state of the DataStore since you retrieved the cookie, then SetChanges will fail.

Examples

These statements use GetChanges to capture changes to a DataWindow control on a client. If GetChanges succeeds, the client calls a remote object function that applies the changes to a DataStore on the server and updates the database:

```
blob lblb_changes
long ll_rv

ll_rv = dw_employee.GetChanges(lblb_changes)

IF ll_rv = -1 THEN
   MessageBox("Error", "GetChanges call failed!")
ELSE
   iuo_employee.UpdateData(lblb_changes)
END IF
```

See also

GetFullState
GetStateStatus
SetChanges
SetFullState

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 and DataStore objects

Syntax

dwcontrol.**GetChild** (*name*, *dwchildvariable*)

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore 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*. If any argument's value is NULL, GetChild returns NULL.

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 using the Name attribute of the Report object's property sheet so that you can refer to it in the GetChild function. You must use the report name (not the name of the DataWindow object in which the report has been placed) when calling GetChild.

Use GetChild when you need to explicitly retrieve data for a child DataWindow or report. Although PowerBuilder automatically retrieves data for the child or report 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).

Nested reports and external data sources
You can't use GetChild to get a reference to a report in a composite DataWindow when the data source of the nested report is external.

Changing property values with the Modify function can cause the reference returned by GetChild to become invalid. After setting such an property, call GetChild again. If a property causes this behavior, it is noted in its description in the *DataWindow Reference*.

Examples

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

GetChildrenList

Description

Provides a list of the children of a routine included in a trace tree model.

Applies to

TraceTreeObject, TraceTreeRoutine, and TraceTreeGarbageCollect objects

Syntax

instancename.**GetChildrenList** (*list*)

Argument	Description
instancename	Instance name of the TraceTreeObject, TraceTreeRoutine, or TraceTreeGarbageCollect object
list	An unbounded array variable of data type TraceTreeNode in which GetChildrenList stores a TraceTreeNode object for each child of a routine. This argument is passed by reference

Return value

ErrorReturn. Returns the following values:

- Success!—The function succeeded
- ModelNotExistsError!—The model does not exist

Usage

You use the GetChildrenList function to extract a list of the children of a routine (the classes and routines it calls) included in a trace tree model. Each child listed is defined as a TraceTreeNode object and provides the type of activity represented by that child.

You must have previously created the trace tree model from a trace file using the BuildModel function.

When the GetChildrenList function is called for TraceTreeGarbageCollect objects, each child listed usually represents the destruction of a garbage collected object.

Examples

This example checks the activity type of a node included in the trace tree model. If the activity type is an occurrence of a routine, it determines the name of the class that contains the routine and provides a list of the classes and routines called by that routine:

```
TraceTree ltct_node
TraceTreeNode ltctn_list
...
CHOOSE CASE node.ActivityType
   CASE ActRoutine!
      TraceTreeRoutine ltctrt_rout
      ltctrt_rout = ltct_node
```

```
                result += "Enter " + ltctrt_rout.ClassName &
                + "." + ltctrt_rout.name + " " &
                + String(ltctrt_rout.ObjectID) + " " &
                + String(ltctrt_rout.EnterTimerValue) &
                + "~r~n" ltctrt_rout.GetChildrenList(ltctn_list)
        ...
```

See also BuildModel

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

dwcontrol.**GetClickedColumn** ( )

Argument	Description
dwcontrol	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 *dwcontrol*. 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). If *dwcontrol* is NULL, GetClickedColumn returns NULL.

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.

Examples

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

Syntax

dwcontrol.**GetClickedRow** ( )

Argument	Description
dwcontrol	The name of the DataWindow control 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 *dwcontrol*. 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). If *dwcontrol* is NULL, GetClickedRow returns NULL.

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.

Not on child DataWindows
The GetClickedRow function does not work on child DataWindows.

Examples

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

Retrieves column information for a DataWindow, child DataWindow, or ListView control.

To retrieve	Use
The number of the current column in a DataWindow or child DataWindow	Syntax 1
The properties of a specified column from a ListView	Syntax 2

Syntax 1

For DataWindows and DataStores

Description

Obtains the number of the current column. The current column is the column that has focus.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetColumn** ( )

Argument	Description
dwcontrol	The name of the DataWindow control DataStore, or child DataWindow for which you want the number of the current column

Return value

Integer. Returns the number of the current column in *dwcontrol*. 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. If *dwcontrol* is NULL, GetColumn returns NULL.

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 (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.

Examples

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

Syntax 2

For ListView controls

Description

Retrieves the properties of a specified column.

Applies to

ListView controls

Syntax

listviewname.**GetColumn** (*index, label, alignment, width*)

Argument	Description
listviewname	The name of the ListView control from which you want to find the properties for a column
index	An integer whose value is the index of the column for which you want to find properties
label	A string identifying the label of the column for which you want to find properties. This argument is passed by reference

Argument	Description
alignment	A value of the enumerated data type Alignment specifying the alignment of the column for which you want to find properties. Values are: ♦ Center! ♦ Justify! ♦ Left! ♦ Right! This argument is passed by reference
width	An integer whose value is the width of the column for which you want to find properties. This argument is passed by reference

Return value

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

Usage

Use *label*, *alignment*, and *width* to retrieve the properties for a specified column.

Examples

This example uses the instance variable li_col to pass the column number to GetColumn and retrieve the properties for the column. The script uses SetColumn to change the column's alignment:

```
string ls_label,ls_align
int li_width
alignment la_align

IF lv_list.View <> ListViewReport! THEN
   lv_list.View = ListViewReport!
END IF

IF li_col = 0 THEN
   MessageBox("Error!","Click on a Column bar.", &
      StopSign!)
ELSE
   lv_list.GetColumn(li_col, ls_label, la_align, &
      li_width)
   lv_list.SetColumn(li_col, ls_label, Right!, &
      li_width)
END IF
```

See also

SetColumn

GetColumnName

Description

Obtains the name of the current column. The current column is the column that has the focus.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetColumnName** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow for which you want the name of the current column

Return value

String. Returns the name of the current column in *dwcontrol*. Returns the empty string ("") if no column is current or if an error occurs. If *dwcontrol* is NULL, GetColumnName returns NULL.

Usage

For information on the current column, see GetColumn.

Examples

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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

GetCommandDDE (*string*)

Argument	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). If *string* is NULL, GetCommandDDE returns NULL.

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 GetCommandDDE to find out what command has been sent. 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.

Examples

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[ ]
```

This script that follows 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

When called by the DDE server application, obtains the application name parameter used by the DDE client sending the command.

Platform information

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

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

GetCommandDDEOrigin (*applstring*)

Argument	Description
applstring	A string variable in which GetCommandDDEOrigin will store the name of the server 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). If *applstring* is NULL, GetCommandDDEOrigin returns NULL.

Usage

The server application calling this function can use the application name (its own DDEname) to determine if it wants to respond to this command. Otherwise, the function provides no additional information about the client.

Examples

This script uses the local variable ls_name to store the name the client application used to identify the server application:

```
string ls_name
GetCommandDDEOrigin(ls_name)
```

See also

GetCommandDDE
StartServerDDE
StopServerDDE

GetCompanyName

Description

Returns the company name for the current execution context.

Applies to

ContextInformation objects

Syntax

servicereference.**GetCompanyName** (*name*)

Argument	Description
servicereference	Reference to the ContextInformation service instance
name	String into which the function places the company name. This argument is passed by reference

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to determine the company name (such as Sybase, Inc.).

Examples

This example calls the GetCompanyName function:

```
String ls_company
Integer li_return
ContextInformation lci_info

li_return = lci_info.GetCompanyName(ls_company)
IF li_return = 1 THEN
   sle_co_name.text = ls_company
END IF
```

See also

GetContextService
GetFixesVersion
GetHostObject
GetMajorVersion
GetMinorVersion
GetName
GetShortName
GetVersionName

GetContextKeywords

Description

Retrieves one or more values associated with a specified keyword.

Applies to

ContextKeyword objects

Syntax

servicereference.**GetContextKeywords** (*name*, *values*)

Argument	Description
servicereference	Reference to the ContextKeyword service instance
name	String specifying the keyword for which the function returns corresponding values
values	Unbounded String array into which the function places the values that correspond to *name*. This argument is passed by reference

Return value

Integer. Returns the number of elements in *values* if the function succeeds and -1 if an error occurs.

Usage

Call this function to access environment variables. Environment-variable availability differs by execution context:

- **PowerBuilder execution time** The function accesses DOS and UNIX environment variables, each of which has a unique keyword
- **PowerBuilder window plug-in** The function accesses keywords specified in the EMBED tag. These keywords need not be unique. If there is no keyword specified in the EMBED tag, the function attempts to access a DOS or UNIX environment variable with the specified name
- **PowerBuilder window ActiveX** The function accesses DOS environment variables, each of which has a unique keyword

On Macintosh
The Macintosh does not use environment variables.

Examples

This example calls the GetContextKeywords function:

```
String ls_keyword
Integer li_count, li_return
ContextKeyword lcx_key

li_return = this.GetContextService &
    ("Keyword", lcx_key)
```

```
ls_keyword = sle_name.Text
lcx_key.GetContextKeywords &
   (ls_keyword, is_values)
FOR li_count = 1 to UpperBound(is_values)
   lb_parms.AddItem(is_values[li_count])
NEXT
```

See also GetContextService

GetContextService

Description

Creates a reference to a context-specific instance of the specified service.

Syntax

GetContextService (*servicename, servicereference*)

Argument	Description
servicename	String specifying the service object. Valid values are: ♦ ContextInformation—Context information service ♦ Internet—Internet service ♦ Keyword—Context keyword service
servicereference	PowerObject into which the function places a reference to the service object specified by *servicename*. This argument is passed by reference

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to establish a reference to a service object, allowing you to access methods and properties in the service object. You must call this function before calling service object functions.

Using a CREATE statement
You can instantiate these objects with a PowerScript CREATE statement. But this always creates an object for the default context (native PowerBuilder execution environment), regardless of where the application is running.

Examples

This example calls the GetContextService function:

```
Integer li_return
ContextKeyword lcx_key

li_return = this.GetContextService &
    ("Keyword", lcx_key)
sle_classname.Text = ClassName(lcx_key)
...
```

See also

GetCompanyName
GetContextKeywords
GetHostObject
GetMajorVersion
GetMinorVersion
GetName
GetShortName
GetURL
GetVersionName
HyperLinkToURL
PostURL

GetData

Obtains data from a control.

To obtain	Use
The value of a data point in a series in a graph	Syntax 1
The unformatted data from an EditMask control	Syntax 2
Data from an OLE server	Syntax 3

Syntax 1

For data points in graphs

Description

Gets 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 and DataStore objects

Syntax

controlname.**GetData** ({ *graphcontrol*, } *seriesnumber*, *datapoint* {, *datatype* })

Argument	Description
controlname	The name of the graph from which you want data, or the name of the DataWindow control or DataStore containing the graph
graphcontrol (DataWindow control or DataStore only) (optional)	A string whose value is the name of the graph from which you want the data when *controlname* is a DataWindow or DataStore
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) (optional)	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

Double. Returns the value of the data in *datapoint* if it succeeds and 0 if an error occurs. If any argument's value is NULL, GetData returns NULL.

Usage

You can use GetData only for graphs whose values axis is numeric. For graphs with other types of values axes, use the GetDataValue function instead.

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!)
```

See also

DeleteData
FindSeries
GetDataValue
InsertData

ObjectAtPointer

Syntax 2 For EditMask controls

Description

Gets the unformatted text from an EditMask control.

Applies to

EditMask controls

Syntax

editmaskname.**GetData** (*datavariable*)

Argument	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, and time

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, GetData returns NULL.

Usage

You can find out the data type of an EditMask control by looking at its MaskDataType property, which holds a value of the MaskDataType enumerated data type.

Examples

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
```

Syntax 3 For data in an OLE server

Description

Gets data from the OLE server associated with an OLE control using Uniform Data Transfer.

Applies to

OLE controls and OLE custom controls

Syntax

olename.**GetData** (*clipboardformat*, *data*)

Argument	Description
olename	The name of the OLE or custom control containing the object you want to populate with data
clipboardformat	The format for the data. You can specify a standard format with a value of the ClipboardFormat enumerated data type. You can specify a nonstandard format as a string Values for *clipboardformat* are: ClipFormatBitmap! ClipFormatDIB! ClipFormatDIF! ClipFormatEnhMetafile! ClipFormatHdrop! ClipFormatLocale! ClipFormatMetafilePict! ClipFormatOEMText! ClipFormatPalette! ClipFormatPenData! ClipFormatRIFF! ClipFormatSYLK! ClipFormatText! ClipFormatTIFF! ClipFormatUnicodeText! ClipFormatWave! If *clipboardformat* is an empty string or a null value, GetData uses the format ClipFormatText!
data	A string or blob variable that will contain the data from the OLE server. If the data you want to get is not appropriate for a string, you must use a blob variable

Return value

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

Usage

GetData will return an error if you specify a clipboard format that the OLE server doesn't support. To find out what formats it supports, see the documentation for the OLE server.

GetData operates via Uniform Data Transfer, a mechanism defined by Microsoft for exchanging data with container applications. PowerBuilder enables data transfer via a global handle. The OLE server must also support data transfer via a global handle. If it does not, you cannot transfer data to or from that server.

Examples

After the user has activated a Microsoft Word document and edited its contents, this example gets the contents from the OLE control ole_word6 and stores the contents in the string ls_oledata. The contents of the string are then displayed in the MultiLineEdit mle_text:

```
string ls_oledata
integer li_rtn

li_rtn = ole_word6.GetData( &
   ClipFormatText!, ls_oledata)
mle_text.Text = ls_oledata
```

One OLE control displays a Microsoft Word document containing a table of data. This example gets the data in the report and assigns it to a graph in a second OLE control. Microsoft Graph in the second control interprets the first row in the table as headings, and subsequent rows as categories or series, depending on the settings on the Data menu:

```
string ls_data
integer li_rtn

li_rtn = ole_word.GetData(ClipFormatText!, ls_data)
IF li_rtn <> 1 THEN RETURN

li_rtn = ole_graph.SetData(ClipFormatText!, ls_data)
```

See also

SetData

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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

GetDataDDE (*string*)

Argument	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). If *string* is NULL, GetDataDDE returns NULL.

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.

Examples

Assuming that your PowerBuilder DDE client application has established a hot link with row 7, column 15 of an Excel spreadsheet, and that the value in that row and column address has changed from red to green (which triggers the HotLinkAlarm event in your application), this script for the HotLinkAlarm event 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
StopServerDDE
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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

GetDataDDEOrigin (*applstring, topicstring, itemstring*)

Argument	Description
applstring	A string variable in which GetDataDDEOrigin will store the name of the server 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). If any argument's value is NULL, GetDataDDEOrigin returns NULL.

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 topic and item requested by the client. The application name is the application specified by the client (the server's own DDEname).

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.

Examples

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 and DataStore objects

Syntax

controlname.**GetDataPieExplode** ({ *graphcontrol*, } *series*, *datapoint*, *percentage*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control or DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore for which you want the percentage a pie slice is exploded
series	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. If any argument's value is NULL, GetDataPieExplode returns NULL.

Examples

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 property Graphite. 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

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	Use
Data point's colors	Syntax 1
Line style and width used by the data point	Syntax 2
Fill pattern or symbol for the data point	Syntax 3

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 properties 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 2-dimensional line graph, but that fill pattern will not be visible.

FOR INFO For the enumerated data type values that GetDataStyle will store in *linestyle* and *enumvariable*, see SetDataStyle.

Syntax 1

For the colors of a data point

Description

Obtains the colors associated with a data point in a graph.

Applies to

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

Syntax

controlname.**GetDataStyle** ({ *graphcontrol*, } *seriesnumber*, *datapointnumber*, *colortype*, *colorvariable*)

Argument	Description
controlname	The name of the graph for which you want the color of a data point, or the name of the DataWindow control or DataStore containing the graph

Argument	Description
graphcontrol (Data Window control and DataStore only) (optional)	When *controlname* is a DataWindow control or DataStore, the name of the graph 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

Integer. Returns 1 if it succeeds and -1 if an error occurs. Stores a color value in *colorvariable*. If any argument's value is NULL, GetDataStyle returns NULL.

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 = gr_emp_data.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)
```

See also

FindSeries
GetSeriesStyle
SetDataStyle
SetSeriesStyle

Syntax 2

For the line style and width used by a data point

Description

Obtains the line style and width for a data point in a graph.

Applies to

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

Syntax

controlname.**GetDataStyle** ({ *graphcontrol*, } *seriesnumber*, *datapointnumber*, *linestyle*, *linewidth*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control only) (optional)	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
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

Argument	Description
linewidth	An integer variable in which you want to store the width of the line. The width is measured in pixels

Return value

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*. If any argument's value is NULL, GetDataStyle returns NULL.

Usage

For the enumerated data type values that GetDataStyle will store in *linestyle*, see SetDataStyle.

Examples

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)
```

See also

FindSeries
GetSeriesStyle
SetDataStyle
SetSeriesStyle

Syntax 3

For the fill pattern or symbol of a data point

Description

Obtains the fill pattern or symbol of a data point in a graph.

Applies to

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

Syntax

controlname.**GetDataStyle** ({ *graphcontrol*, } *seriesnumber*, *datapointnumber*, *enumvariable*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control or DataStore only) (optional)	A string whose value is the name of the graph (in the DataWindow control or DataStore) 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

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. If any argument's value is NULL, GetDataStyle returns NULL.

Usage

For the enumerated data type values that GetDataStyle will store in *enumvariable*, see SetDataStyle.

Examples

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 a data point in the variable symbol_type. The data point is the sixth point in the series named in the SingleLineEdit sle_series in the graph gr_depts in the DataWindow control dw_employees:

```
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 and DataStore objects

Syntax

controlname.**GetDataValue** ({ *graphcontrol*, } *seriesnumber*, *datapoint*, *datavariable* {, *xory* })

Argument	Description
controlname	The name of the graph from which you want data, or the name of the DataWindow control or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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) (optional)	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. If any argument's value is NULL, GetDataValue returns NULL.

Usage

GetDataValue retrieves data from any graph. The data is stored in *datavariable*, whose data type must match the data type of the graph's values axis. If the values axis is numeric, you can also use the GetData function.

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.GetDataValue( &
     "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.GetDataValue( &
       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.GetDataValue( &
       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.

Restriction
You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicDate** (*index*)

Argument	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

Date. Returns the Date data in the output parameter descriptor identified by *index* in *DynamicDescriptionArea*. Returns 1900-01-01 if an error occurs. If any argument's value is NULL, GetDynamicDate returns NULL.

Usage

After you fetch data using Format 4 dynamic SQL statements, the DynamicDescriptionArea, usually SQLDA, contains information about the data retrieved. The SQLDA property NumOutputs specifies the number of data descriptors returned. The property 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 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

GetDynamicDateTime
GetDynamicNumber
GetDynamicString
GetDynamicTime
SetDynamicParm
Using dynamic SQL

GetDynamicDateTime

Description

Obtains data of type DateTime from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

Restriction
You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicDateTime** (*index*)

Argument	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. If any argument's value is NULL, GetDynamicDateTime returns NULL.

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.

Examples

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

GetDynamicDate
GetDynamicNumber
GetDynamicString
GetDynamicTime
SetDynamicParm
Using dynamic SQL

GetDynamicNumber

Description

Obtains numeric data from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

Restriction
You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicNumber** (*index*)

Argument	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. If any argument's value is NULL, GetDynamicNumber returns NULL.

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.

Examples

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

GetDynamicDate
GetDynamicDateTime
GetDynamicString
GetDynamicTime
SetDynamicParm
Using dynamic SQL

GetDynamicString

Description

Obtains data of type String from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

Restriction
You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicString** (*index*)

Argument	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. If any argument's value is NULL, GetDynamicString returns NULL.

Usage

Use GetDynamicString when the value of OutParmType is TypeString! for the value that you want to retrieve from the array.

Examples

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

GetDynamicDate
GetDynamicDateTime
GetDynamicNumber
GetDynamicTime
SetDynamicParm
Using dynamic SQL

GetDynamicTime

Description

Obtains data of type Time from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

Restriction
You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicTime** (*index*)

Argument	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. If any argument's value is NULL, GetDynamicTime returns NULL.

Usage

Use GetDynamicTime when the value of OutParmType is TypeTime! for the value that you want to retrieve from the array.

Examples

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

GetDynamicDate
GetDynamicDateTime
GetDynamicNumber
GetDynamicString
SetDynamicParm
Using dynamic SQL

GetEnvironment

Description

Gets information about the operating system, processor, and screen display of the system.

Syntax

GetEnvironment (*environmentinfo*)

Argument	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. If *environmentinfo* is NULL, GetEnvironment returns NULL.

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 properties.

Examples

This script runs another PowerBuilder application and uses the OSType property 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:PB Apps Folder:Analyze"
CASE aix!
   path = "/export/home/pb_apps/analyze.exe"
CASE Windows!, WindowsNT!
   path = "C:\PB_apps\analyze.exe"
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* } })

Argument	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. If any argument's value is NULL, GetFileOpenName returns NULL.

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"
```

Opening a file
Use the FileOpen function to open a selected file.

The dialog boxes presented by GetFileOpenName and GetFileSaveName are system dialog boxes. They provide standard system behavior, including control over the current directory. When users change the drive, directory, or folder in the dialog box, they change the current directory or folder. The newly selected directory or folder becomes the default for file operations until they exit the application.

Platform information
On Macintosh, the function processes filenames, extensions, and filters as it does in Windows and UNIX.

Examples

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* } })

Argument	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
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. If any argument's value is NULL, GetFileSaveName returns NULL.

Usage

For usage notes on the *filter* argument and the current directory, see the GetFileOpenName function.

Platform information

On the Macintosh, the function processes filenames, extensions, and filters as it does in Windows.

Examples

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 ( )**

Argument	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. If *mdiframewindow* is NULL, GetFirstSheet returns NULL.

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.

Examples

This script for a menu selection 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

GetFixesVersion

Description

Returns the fix level for the current PowerBuilder execution context. For example, at maintenance level 6.0.03, the fix version is 03.

Applies to

ContextInformation objects

Syntax

servicereference.**GetFixesVersion** (*fixversion*)

Argument	Description
servicereference	Reference to the ContextInformation service instance
fixversion	Integer into which the function places the fix version. This argument is passed by reference

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to determine the current fix version.

Examples

This example calls the GetFixesVersion function:

```
String ls_name
Constant String ls_currver = "6.0.02"
Integer li_majver, li_minver, li_fixver
ContextInformation lci_info

this.GetContextService &
   ("ContextInformation", lci_info)
lci_info.GetMajorVersion(li_majver)
lci_info.GetMinorVersion(li_minver)
lci_info.GetFixesVersion(li_fixver)
IF li_majver <> 6 THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
ELSEIF li_minver <> 0 THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
ELSEIF li_fixver <> 2 THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
END IF
```

See also

GetCompanyName
GetHostObject
GetMajorVersion
GetMinorVersion
GetName
GetShortName
GetVersionName

GetFocus

Description

Determines the control that currently has focus.

Syntax

GetFocus ( )

Return value

GraphicObject. Returns the control that currently has focus. Returns an invalid control reference if an error occurs.

Use the IsValid function to determine whether GetFocus has returned a valid control.

Examples

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 property 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
```

See also

IsValid
SetFocus

GetFormat

Description

Obtains the display format assigned to a column in a DataWindow control or DataStore object.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetFormat** (*column*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol*. If an error occurs, GetFormat returns the empty string (""). If any argument's value is NULL, GetFormat returns NULL.

Usage

If you want to temporarily change the display format of a column, you can use GetFormat to save the current format.

Examples

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

GetFullState

Description

Retrieves the complete state of a DataWindow or DataStore into a blob.

This function is used primarily in distributed applications.

Applies to

DataWindow controls and DataStore objects

Syntax

dwcontrol.**GetFullState** (*dwasblob*)

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore for which you want to retrieve state information
dwasblob	A blob into which the returned DataWindow will be placed

Return value

Long. Returns the number of rows in the DataWindow blob if it succeeds and -1 if an error occurs. GetFullState will return -1 if the DataWindow control or DataStore does not have a DataWindow object associated with it.

Usage

GetFullState retrieves the entire state of a DataWindow or DataStore into a blob, including the DataWindow object specification, the data buffers, and the status flags. When you set SetFullState to apply the blob created by GetFullState to another DataWindow, the target DataWindow has enough information to recreate the source DataWindow.

Because the blob created by GetFullState contains the DataWindow object specification, a subsequent call to SetFullState will overwrite the DataWindow object for the target DataWindow control or DataStore. If the target of SetFullState does not have a DataWindow object associated with it, the blob will assign one. In this case, SetFullState has the effect of setting the DataObject property for the target.

When you use GetFullState and SetFullState to synchronize a DataWindow control on a client with a DataStore on a server, you need to make sure that the DataWindow object for the DataStore contains the presentation style you want to display on the client.

Examples

These statements retrieve data into a DataStore and use GetFullState to retrieve the complete state of the DataStore into a blob:

```
// Instance variable:
// datastore ids_datastore

long ll_rv
```

```
ids_datastore = create datastore
ids_datastore.dataobject = "d_emplist"
ids_datastore.SetTransObject (SQLCA)

ids_datastore.Retrieve()

ll_rv = ids_datastore.GetFullState(ablb_data)
```

See also

GetChanges
GetStateStatus
SetChanges
SetFullState

GetHostObject

Description

Provides a reference to the context's host object.

Host object support
Currently, host object support is implemented only in the window ActiveX when running under Internet Explorer. In this situation GetHostObject returns a reference to the IWebBrowserApp ActiveX automation server object.

Applies to

ContextInformation objects

Syntax

servicereference.**GetHostObject** (*hostobject*)

Argument	Description
servicereference	Reference to the Context Information service instance
hostobject	PowerObject into which the function places a reference to the ActiveX automation server object

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to obtain a reference to the context object model. If running the window ActiveX under Internet Explorer 3.0 or greater and *hostobject* is an uninstantiated OleObject variable, the function returns a reference to an ActiveX automation server object, which you can use to control the hosting browser. If host object support is not available, the function returns -1 and *hostobject* is NULL.

Examples

This example calls the GetHostObject function. Ici_info is an instance variable of type ContextInformation, which has been populated via the GetContextService function; ole1 is an instance variable of type OLEObject:

```
Integer li_return

li_return = ici_info.GetHostObject(ole1)
IF li_return = 1 THEN
   sle_1.Text = "GetHostObject succeeded"
ELSE
   sle_1.Text = "GetHostObject failed"
   cb_goback.Enabled = FALSE
   cb_navigate.Enabled = FALSE
END IF
```

See also

GetCompanyName
GetFixesVersion
GetMajorVersion
GetMinorVersion
GetName
GetShortName
GetVersionName

GetItem

Retrieves data associated with a specified item in ListView and TreeView controls.

To retrieve data associated with a specified	Use
Column in a ListView control	Syntax 1
ListView item	Syntax 2
TreeView item	Syntax 3

Syntax 1

For ListView controls

Description

Retrieves the label for a specified column.

Applies to

ListView controls

Syntax

listviewname.**GetItem** (*index, column, label*)

Argument	Description
listviewname	The name of the ListView control from which you want to retrieve a column label
index	The index number of the item for which you want to retrieve the label
column	The column number of the column for which you want to retrieve the label
label	A variable for the label of the column you want to retrieve

Return value

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

Usage

This syntax is required to display a report view. If you are not retrieving column information from a report view, use Syntax 2.

Examples

This example uses retrieves column information that is used in the SetItem function:

```
string ls_artist, ls_comp

lv_list.GetItem(1, 3 , ls_artist)
lv_list.GetItem(1, 1 , ls_comp)
```

```
sle_info.text = ls_artist +" wrote " + ls_comp + "."

lv_list.SetItem(11 , 1 , "Blue Seven")
lv_list.SetItem(11 , 2, "Saxophone Colossus")
lv_list.SetItem(11 , 3 , ls_artist)
```

See also SetItem

Syntax 2 For ListView controls

Description Retrieves the state information for a specified ListView item.

Applies to ListView controls

Syntax *listviewname*.**GetItem** (*index, item*)

Argument	Description
listviewname	The name of the ListView control for which you want to retrieve state information
index	The index number of the item for which you want to retrieve state information
item	A ListViewItem variable in which you want to store the item identified by *index*

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

Usage If you are retrieving column information from a report view, use syntax 1.

Examples This example uses GetItem to move second item in a ListView to the fifth item. It retrieves the state information for item 2, inserts it into the ListView control as item 5, and then deletes the original item:

```
listviewitem l_lvi

lv_list.GetItem(2 , l_lvi)
lv_list.InsertItem(5 , l_lvi)
lv_list.DeleteItem(2)
```

See also SetItem

Syntax 3

For TreeView controls

Description

Retrieves the data associated with the specified item.

Applies to

TreeView controls

Syntax

treeviewname.**GetItem** (*itemhandle, item*)

Argument	Description
treeviewname	The name of the TreeView control in which you want to get data for a specified item
itemhandle	The handle for the item for which you want to retrieve information
item	A TreeViewItem variable in which you want to store the item identified by the item handle

Return value

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

Usage

Use GetItem to retrieve the state information associated with a specific TreeView item (such as label, handle, picture index). Once you have retrieved the information, you can use it in your application.

To change a property of an item in a TreeView, call GetItem to assign the item to a TreeViewItem variable, change its properties, and call SetItem to copy the changes back to the TreeView.

Examples

This code for the Clicked event gets the clicked item and changes it overlay picture. The SetItem function copies the change back to the TreeView:

```
treeviewitem tvi
This.GetItem(handle, tvi)
tvi.OverlayPictureIndex = 1
This.SetItem(handle, tvi)
```

This example tracks items in the SelectionChanged event. If there is no prior selection, the value of l_tviold is zero:

```
treeviewitem l_tvinew, l_tviold

// Get the treeview item that was the old selection
tv_list.GetItem(oldhandle, l_tviold)

// Get the treeview item that is currently selected
tv_list.GetItem(newhandle, l_tvinew)
```

```
// Print the labels for the two items in the
// SingleLineEdit
sle_get.Text = "Selection changed from " &
   + String(l_tviold.Label) + " to " &
   + String(l_tvinew.Label)
```

See also InsertItem

GetItemDate

Description

Gets data whose type is Date from the specified buffer of a DataWindow control or DataStore object. 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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetItemDate** (*row*, *column* {, *dwbuffer*, *originalvalue* })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 (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 (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. If any argument's value is NULL, GetItemDate returns NULL.

Usage

Use GetItemDate when you want to get information from the DataWindow's buffers. To find out what the user entered in the current column before that data is accepted, use GetText. In the ItemChanged or ItemError events, use the data argument.

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.

Data types of columns and computed fields

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.

There is a difference in data types between columns and computed columns retrieved from the database and computed fields defined in the DataWindow painter. Computed columns from the database can have a data type of date, but a date computed field always has a data type of DateTime, not date. Use the GetItemDateTime function instead.

Using GetItemDate in a String function

When you call GetItemDate as an argument for the String function and don't specify a display format, the value is formatted as a DateTime value. This statement returns a string like "2/26/96 00:00:00":

```
String(dw_1.GetItemDate(1, "start_date"))
```

To get a simple date string, you can specify a display format:

```
String(dw_1.GetItemDate(1,"start_date"), "m/d/yy")
```

or you can assign the date to a date variable before calling the String function:

```
date ld_date
string ls_date
ld_date = dw_1.GetItemDate(1, "start_date")
ls_date = String(ld_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 or DataStore object. 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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetItemDateTime** (*row, column* {, *dwbuffer, originalvalue* })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 (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 specific *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. If any argument's value is NULL, GetItemDateTime returns NULL.

Usage

Use GetItemDateTime when you want to get information from the DataWindow's buffers. To find out what the user entered in the current column before that data is accepted, use GetText. In the ItemChanged or ItemError events, use the data argument.

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.

Data type mismatch
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.

Computed fields displaying date or time values have a data type of DateTime, not date or time. Always use GetItemDateTime to get their value, not GetItemDate or GetItemTime.

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 or DataStore object. 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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetItemDecimal** (*row, column* {, *dwbuffer, originalvalue* })

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore 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). 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 (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. Triggers the SystemError event and returns -1 if an error occurs (see Handling errors below). If any argument's value is NULL, GetItemDecimal returns NULL.

Usage

Use GetItemDecimal when you want to get information from the DataWindow's buffers. To find out what the user entered in the current column before that data is accepted, use GetText. In the ItemChanged or ItemError events, use the data argument.

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.

Handling errors

The return value is a valid value from the database unless the SystemError event is triggered. When the value cannot be converted because the column's data type does not match the function's data type, an execution error occurs, which triggers the SystemError event. The default error processing halts the application. If you write a script for the SystemError event, it should also halt the application. Therefore, the error return value is seldom used.

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 or DataStore object. 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

dwcontrol.**GetItemNumber** (*row, column* {, *dwbuffer, originalvalue* })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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). 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 (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 (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. Triggers the SystemError event and returns -1 if an error occurs (see *Handling errors* below). If any argument's value is NULL, GetItemNumber returns NULL.

Usage

Use GetItemNumber to get information from the DataWindow's buffers. To find out what the user entered in the current column before that data is accepted, use GetText. In the ItemChanged or ItemError events, use the data argument.

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.

Handling errors

The return value is a valid value from the database unless the SystemError event is triggered. When the value cannot be converted because the column's data type does not match the function's data type, an execution error occurs, which triggers the SystemError event. The default error processing halts the application. If you write a script for the SystemError event, it should also halt the application. Therefore, the error return value is seldom used.

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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetItemStatus** (*row, column, dwbuffer*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 (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 *dwcontrol* in *dwbuffer*. If column is 0, GetItemStatus returns the status of *row*. If any argument's value is NULL, GetItemStatus returns NULL.

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 rows 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 whose status was New! or NewModified! before being moved to the delete buffer.

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 or DataStore object. 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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetItemString** (*row, column* {, *dwbuffer, originalvalue* })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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). 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 (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 (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. If any argument's value is NULL, GetItemString returns NULL.

Usage

Use GetItemString to get information from the DataWindow's buffers. To find out what the user entered in the current column before that data is accepted, use GetText. In the ItemChanged or ItemError events, use the data argument.

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.

Mismatched data types
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 or DataStore object. 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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetItemTime** (*row*, *column* {, *dwbuffer*, *originalvalue* })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 be time. *Column* can be a column number (integer) or a column name (string). 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 (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 (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. If any argument's value is NULL, GetItemTime returns NULL.

Usage

Use GetItemTime to get information from the DataWindow's buffers. To find out what the user entered in the current column before that data is accepted, use GetText. In the ItemChanged or ItemError events, use the data argument.

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.

Data types of columns and computed fields

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.

There is a difference in data types between computed columns retrieved from the database and computed fields defined in the DataWindow painter. Computed columns from the database can have a data type of time, but a time computed field always has a data type of DateTime, not time. Use the GetItemDateTime function instead.

Using GetItemTime in a String function

When you call GetItemTime as an argument for the String function and don't specify a display format, the value is formatted as a DateTime value. This statement returns a string like "2/26/96 00:00:00":

```
String(dw_1.GetItemTime(1, "start_date"))
```

To get a simple time string, you can specify a display format for the String function or you can assign the value to a time variable before calling the String function (see GetItemDate for examples).

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

GetLastReturn

Description

Returns the return value from the last InvokePBFunction or TriggerPBEvent function.

Applies to

Window ActiveX controls

Syntax

activexcontrol.**GetLastReturn** ( )

Argument	Description
activexcontrol	Identifier for the instance of the PowerBuilder window ActiveX control. When used in HTML, the ActiveX control is the NAME attribute of the object element. When used in other environments, this references the control that contains the PowerBuilder window ActiveX

Return value

Any. Returns the last return value.

Usage

Call this function after calling InvokePBFunction or TriggerPBEvent to access the return value.

JavaScript scripts must use this function to access return values from InvokePBFunction and TriggerPBEvent. VBScript scripts can either use this function or access the return value via an argument in InvokePBFunction or TriggerPBEvent.

Examples

This JavaScript example calls the GetLastReturn function:

```
...
   retcd = PBRX1.TriggerPBEvent(theEvent, numargs);
      rc = parseInt(PBRX1.GetLastReturn());
if (rc != 1) {
      alert("Error. Empty string.");
   }
...
```

This VBScript example calls the GetLastReturn function:

```
...
   retcd = PBRX1.TriggerPBEvent(theEvent, &
      numargs, args)
   rc = PBRX1.GetLastReturn()
   IF rc <> 1 THEN
      msgbox "Error. Empty string."
   END IF
...
```

See also

GetArgElement
InvokePBFunction
SetArgElement
TriggerPBEvent

GetMajorVersion

Description

Returns the major version for the current PowerBuilder execution context. For example, at maintenance level 6.0.03 the major version is 6.

Applies to

ContextInformation objects

Syntax

servicereference.**GetMajorVersion** (*majorversion*)

Argument	Description
servicereference	Reference to the ContextInformation service instance
majorversion	Integer into which the function places the major version. This argument is passed by reference

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to determine the current major version.

Examples

This example calls the GetMajorVersion function:

```
String ls_name
Constant String ls_currver = "6.0.02"
Integer li_majver, li_minver, li_fixver
ContextInformation lci_info

this.GetContextService &
   ("ContextInformation", lci_info)

GetMajorVersion(li_majver)
lci_info.GetMinorVersion(li_minver)
lci_info.GetFixesVersion(li_fixver)
IF li_majver <> 6 THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
ELSEIF li_minver <> 0 THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
ELSEIF li_fixver <> 2 THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
END IF
```

See also

GetCompanyName
GetFixesVersion
GetHostObject
GetMinorVersion
GetName
GetShortName
GetVersionName

GetMessageText

Description

Obtains the message text generated by a crosstab DataWindow object in a DataWindow control. Only crosstab DataWindows generate messages.

Obsolete function
GetMessageText is obsolete and will be discontinued in the near future. You should replace all use of GetMessageText as soon as possible. The message text is available as an argument in a user event defined for pbm_dwnmessagetext in a DataWindow control.

Applies to

DataWindow controls

Syntax

dwcontrol.**GetMessageText** ( )

Argument	Description
dwcontrol	The name of the DataWindow control for which you want the message text

Return value

String. Returns the text of the message generated by *dwcontrol*. If there is no text or an error occurs, GetMessageText returns the empty string (""). If *dwcontrol* is NULL, GetMessageText returns NULL.

Usage

To use GetMessageText, you must first define a user-defined event for the event ID pbm_dwnmessagetext; then you call this function in the script for that event.

Typical messages are Retrieving data and Building crosstab.

Examples

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())
```

GetMinorVersion

Description

Returns the minor version for the current PowerBuilder execution context. For example, at maintenance level 6.0.03 the minor version is 0.

Applies to

ContextInformation objects

Syntax

servicereference.**GetMinorVersion** (*minorversion*)

Argument	Description
servicereference	Reference to the ContextInformation service instance
minorversion	Integer into which the function places the minor version. This argument is passed by reference

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to determine the current minor version.

Examples

This example calls the GetMinorVersion function:

```
String ls_name
Constant String ls_currver = "6.0.02"
Integer li_majver, li_minver, li_fixver
ContextInformation lci_info

this.GetContextService &
   ("ContextInformation", lci_info)

lci_info.GetMajorVersion(li_majver)
lci_info.GetMinorVersion(li_minver)
lci_info.GetFixesVersion(li_fixver)
IF li_majver <> 6 THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
ELSEIF li_minver <> 0 THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
ELSEIF li_fixver <> 2 THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
END IF
```

See also

GetCompanyName
GetFixesVersion
GetHostObject
GetMajorVersion
GetName
GetShortName
GetVersionName

GetName

Description

Gets the name for the current execution context.

Applies to

ContextInformation objects

Syntax

servicereference.**GetName** (*name*)

Argument	Description
servicereference	Reference to the ContextInformation service instance
name	String into which the function places the name. This argument is passed by reference

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs. The function returns values as follows:

- **PowerBuilder execution-time:** PowerBuilder Runtime
- **PowerBuilder window plug-in:** PowerBuilder window Plug-in
- **PowerBuilder window ActiveX:** PowerBuilder Runtime ActiveX

Usage

Call this function to determine the current execution environment.

Examples

This example calls the GetName function. Ici_info is an instance variable of type ContextInformation:

```
String ls_name

this.GetContextService("ContextInformation", &
   ici_info)
ici_info.GetName(ls_name)
IF ls_name <> "PowerBuilder Runtime" THEN
   cb_close.visible = FALSE
END IF
```

See also

GetCompanyName
GetContextService
GetFixesVersion
GetHostObject
GetMajorVersion
GetMinorVersion
GetShortName
GetVersionName

GetNativePointer

Description

Gets a pointer to the OLE object associated with the OLE control. The pointer lets you call OLE functions in an external DLL for the object.

Applies to

OLE controls and OLE custom controls

Syntax

olename.**GetNativePointer** (*pointer*)

Argument	Description
olename	The name of the OLE control containing the object for which you want the native pointer
pointer	A UnsignedLong variable in which you want to store the pointer. If GetNativePointer cannot get a valid pointer, *pointer* is set to 0

Return value

Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage

Using the pointer in your own DLL calls *Pointer* is a pointer to OLE's IUnknown interface. You can use it with QueryInterface to get other interface pointers.

When you call GetNativePointer, PowerBuilder calls OLE's AddRef function, which locks the pointer. You must release the pointer in your DLL function or in a PowerBuilder script with the ReleaseNativePointer function.

Examples

This example gets a pointer for the OLECustomControl ocx_spell for making external function calls for OLE automation:

```
UnsignedLong lul_oleptr
integer li_rtn

li_rtn = ocx_spell.GetNativePointer(lul_oleptr)
IF li_rtn = 0 THEN
    ... // Call external functions for automation
    ocx_spell.ReleaseNativePointer(lul_oleptr)
END IF
```

See also

GetAutomationNativePointer
ReleaseAutomationNativePointer
ReleaseNativePointer

GetNextModified

Description

Reports the next row that has been modified in the specified buffer.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetNextModified** (*row*, *dwbuffer*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 (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 *dwcontrol*. Returns 0 if there are no modified rows after the specified row. If any argument's value is NULL, GetNextModified returns NULL.

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.

Examples

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.")
```

Total number of modified rows

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*)

Argument	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. If any argument's value is NULL, GetNextSheet returns NULL.

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.

Examples

The following script for a menu selection 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

dwcontrol.**GetObjectAtPointer** ( )

Argument	Description
dwcontrol	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. If *dwcontrol* is NULL, GetObjectAtPointer returns NULL.

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 INFO For information on the rows associated with bands and therefore with objects in those bands, see GetBandAtPointer.

Examples

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

GetOrigin

Description

Finds the X and Y coordinates of the upper-left corner of the ListView item.

Applies to

ListView controls

Syntax

listviewname.**GetOrigin** (*x* , *y*)

Argument	Description
listviewname	The ListView control for which you want to find the coordinates of the upper-left corner
x	An integer variable in which you want to store the X coordinate for the ListView control
y	An integer variable in which you want to store the Y coordinate for the ListView control

Return value

Integer. Returns 1 if it succeeds and – 1 if it fails.

Usage

Use GetOrigin to find the position of a dragged object relative to the upper left corner of a ListView control.

Examples

This example moves a static text clock to the upper-left coordinates of the selected ListView item:

```
integer li_index
listviewitem l_lvi

li_index = lv_list.SelectedIndex()
lv_list.GetItem(li_index, l_lvi)

lv_list.GetOrigin(l_lvi.ItemX, l_lvi.ItemY)

sle_info.Text = "X is "+ String(l_lvi.ItemX) &
+ " and Y is " + String(l_lvi.ItemY)

st_clock.Move(l_lvi.itemx , l_lvi.ItemY)

MessageBox("Clock Location", "X is " &
    + String(st_clock.X) &
    + ", and Y is " &
    + String(st_clock.Y)+".")
```

GetParagraphSetting

Description

Gets the size of the indentation, left margin, or right margin of the paragraph containing the insertion point in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtecontrol.**GetParagraphSetting** (*whichsetting*)

Argument	Description
rtecontrol	The name of the control for which you want paragraph information
whichsetting	A value of the ParagraphSetting enumerated data type specifying the setting for which you want the value. Values are: ♦ Indent! — Returns the indentation of the paragraph ♦ LeftMargin! — Returns the left margin of the paragraph ♦ RightMargin! — Returns the right margin of the paragraph

Return value

Long. Returns the size of the specified setting in thousandths of an inch. GetParagraphSetting returns -1 if an error occurs. If *whichsetting* is NULL, it returns NULL.

Examples

This example gets the indentation setting for the current paragraph:

```
long ll_indent
ll_indent = rte_1.GetParagraphSetting(Indent!))
```

See also

GetAlignment
GetSpacing
GetTextColor
GetTextStyle
SetParagraphSetting

GetParent

Description

Obtains the parent of the specified object.

Applies to

Any object

Syntax

objectname.**GetParent** ( )

Argument	Description
objectname	A control in a window or user object or an item on a menu for which you want the parent object

Return value

PowerObject. Returns a reference to the parent of *objectname*.

Examples

In event scripts for a user object that will be used as a tab page, you can use code like the following to make references to the parent Tab control generic:

```
// a_tab is generic;
// it doesn't know about specific pages
tab a_tab

// a_tab_page is generic;
// it doesn't know about specific controls
userobject a_tab_page

// Get values for the Tab control and the tab page
a_tab = this.GetParent( )
// Somewhat redundant, for illustration only
a_tab_page = this

// Set properties for the tab page
a_tab_page.PowerTipText = "Important property page"
// Set properties for the Tab control
a_tab.PowerTips = TRUE

// Run Tab control functions
a_tab.SelectTab(a_tab_page)
```

You can't refer to controls on the user object because a_tab_page doesn't know about them. You can't refer to specific pages in the Tab control because a_tab doesn't know about them either.

In event scripts for controls on the tab page user object, you can use two levels of GetParent to refer to the user object and the Tab control containing the user object as a tab page:

```
// For a control, add one more level of GetParent()
// and you can make the same settings as above
tab a_tab
userobject a_tab_page

a_tab_page = this.GetParent()
a_tab = a_tab_page.GetParent()

a_tab_page.PowerTipText = "Important property page"
a_tab.PowerTips = TRUE

a_tab.SelectTab(a_tab_page)
```

See also

ParentWindow
"Pronouns" on page 14

GetRemote

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.

To	Use
Make a single request of a DDE server application (called a cold link)	Syntax 1
Request data from a DDE server application after you have opened a channel (called a warm link)	Syntax 2

Syntax 1

For single DDE requests

Description

Asks a DDE server application to provide data and stores that data in the specified variable without requiring an open channel. This syntax is appropriate when you will make only one or two requests of the server.

Platform information

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

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

GetRemote (*location, target, applname, topicname*)

Argument	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

Argument	Description
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

Integer. Returns 1 if it succeeds and a negative integer if an error occurs. Values are:

-1 Link was not started
-2 Request denied

If any argument's value is NULL, GetRemote returns NULL.

Usage

When using DDE, your PowerBuilder application must have an open window, which will be the client window. For this syntax, the active window is the DDE client window.

FOR INFO For more information about DDE channels and warm and cold links, see the two syntaxes of 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. 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")
```

See also

ExecRemote
SetRemote

Syntax 2

For DDE requests via an open channel

Description

Asks a DDE server application to provide data and stores that data in the specified variable when you have already established a warm link by opening a channel to the server. 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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

GetRemote (*location, target, handle* {, *windowhandle* })

Argument	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

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 this syntax, you can specify the client window with the *windowhandle* argument.

Before using this syntax, call OpenChannel to establish a DDE channel.

FOR INFO For more information about DDE channels and warm and cold links, see the ExecRemote function.

Examples

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. 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 or DataStore object.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetRow** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol*. Returns 0 if no row is current and -1 if an error occurs. If *dwcontrol* is NULL, GetRow returns NULL.

Current row not always displayed
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 scrollbar to scroll to row 50, the current row remains row 7 unless the user clicks row 50.

Examples

This statement returns the number of the current row in dw_Employee:

```
dw_employee.GetRow()
```

See also

GetColumn
SetColumn
SetRow
GetRow in the *DataWindow Reference*

GetRowFromRowId

Description

Gets the row number of a row in a DataWindow control or DataStore object from the unique row identifier associated with that row.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetRowFromRowId** (*rowid* {, *buffer* })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow for which you want the row number
rowid	A long identifying the row identifier for which you want the associated row number
buffer (optional)	A value of the dwBuffer enumerated data type indicating the DataWindow buffer from which you want to get the data. Valid values are: ♦ Primary! — (Default) The data in the primary buffer (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

Long. Returns the row number in *buffer*. Returns 0 if the row number is not in the current buffer and -1 if an error occurs.

Usage

This function allows you to use a unique row identifier to retrieve the associated DataWindow or DataStore row number. The row identifier is not affected by operations (such as Insert, Delete, or Filter) that may change the original order (and consequently the row numbers) of the rows in the DataWindow or DataStore.

Row identifiers
The row identifier is relative to the DataWindow that currently owns the row.

Examples

This example uses the row identifier previously obtained using the GetRowIDFromRow function to retrieve the row's number after the original order of the rows in the DataWindow has changed.

```
long ll_rowid
```

```
long ll_rownumber

ll_rowid = dw_emp.GetRowIDFromRow(dw_emp.GetRow())
// original order of rows changes...
ll_rownumber = dw_emp.GetRowFromRowID(ll_rowid)
```

See also

GetRow
GetRowIdFromRow
GetRow in the *DataWindow Reference*

GetRowIdFromRow

Description

Gets the unique row identifier of a row in a DataWindow control or DataStore object from the row number associated with that row.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetRowIdFromRow** (*rownumber* {, *buffer* })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or the child DataWindow for which you want the row identifier
rownumber	A long identifying the row number for which you want the associated row identifier
buffer (optional)	A value of the dwBuffer enumerated data type indicating the DataWindow buffer from which you want to get the data. Valid values are: ♦ Primary! — (Default) The data in the primary buffer (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

Long. Returns the row identifier in *buffer*. Returns 0 if the row identifier is not in the current buffer and -1 if an error occurs.

Usage

The row identifier value is not the same as the row number value used in many DataWindow and DataStore function calls and should not be used for the row number value. Instead you should first convert the unique row identifier into a row number by calling GetRowFromRowId.

Row identifiers
The row identifier is relative to the DataWindow that currently owns the row.

Examples

This example retrieves a row's unique identifier using the row number obtained with GetRow.

```
long ll_rowid

ll_rowid = dw_emp.GetRowIDFromRow(dw_emp.GetRow())
```

See also

GetRow
GetRowFromRowId
GetRow in the *DataWindow Reference*

GetSelectedRow

Description

Reports the number of the next highlighted row after a specified row in a DataWindow control or DataStore object.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetSelectedRow** (*row*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol*. Returns 0 if no row is selected after the specified row. If any argument's value is NULL, GetSelectedRow returns NULL.

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

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, depending on what settings you want.

To	Use
Get the series' colors	Syntax 1
Get the line style and width used by the series	Syntax 2
Get the fill pattern or symbol for the series	Syntax 3
Find out if the series is an overlay (a series shown as a line on top of another graph type)	Syntax 4

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 to 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 properties 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.

Syntax 1

For the colors of a series

Description

Obtains the colors associated with a series in a graph.

Applies to

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

Syntax

controlname.**GetSeriesStyle** ({ *graphcontrol,* } *seriesname, colortype, colorvariable*)

Argument	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 or DataStore containing the graph

Argument	Description
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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 3-dimensional or have solid data markers)
colorvariable	A long variable in which you want to store the color's RGB value

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. Stores in *colorvariable* the RGB value of the specified series and item. If any argument's value is NULL, GetSeriesStyle returns NULL.

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
```

See also

AddSeries
GetDataStyle
FindSeries
SetDataStyle
SetSeriesStyle

Syntax 2

For the line style and width used by a series

Description

Obtains the line style and width for a series in a graph.

Applies to

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

Syntax

controlname.**GetSeriesStyle** ({ *graphcontrol*, } *seriesname*, *linestyle*, *linewidth*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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

Argument	Description
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

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. If any argument's value is NULL, GetSeriesStyle returns NULL.

Examples

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
```

See also

AddSeries
GetDataStyle
FindSeries
SetDataStyle
SetSeriesStyle

Syntax 3 For the fill pattern or symbol of a series

Description

Obtains the fill pattern or symbol of a series in a graph.

Applies to

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

Syntax

controlname.**GetSeriesStyle** ({ *graphcontrol*, } *seriesname*, *enumvariable*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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

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. If any argument's value is NULL, GetSeriesStyle returns NULL.

Usage

See SetSeriesStyle for a list of the enumerated data type values that GetSeriesStyle stores in *enumvariable*.

Examples

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 these examples, you can change the data type of data_pattern (the variable specified as the last argument) to find out the symbol type.

See also

AddSeries
GetDataStyle
FindSeries
SetDataStyle
SetSeriesStyle

Syntax 4 For determining whether a series is an overlay

Description

Reports whether a series in a graph is an overlay—whether it is shown as a line on top of another graph type.

Applies to

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

Syntax

controlname.**GetSeriesStyle** ({ *graphcontrol*, } *seriesname*, *overlayindicator*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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

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. If any argument's value is NULL, GetSeriesStyle returns NULL.

Examples

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)
```

GetServerInfo

Description

Allows a client application to retrieve information about its connection to a server. GetServerInfo allows a client with administrative privileges to retrieve information about all clients connected to a particular server.

This function applies to distributed applications only.

Applies to

Connection objects

Syntax

connection.**GetServerInfo** (*connectioninfo*)

Argument	Description
connection	The name of the Connection object you want to use to establish the connection. The Connection object must contain the information required for the connection.
connectioninfo	The name of an unbounded array of structures of type ConnectionInfo.

Return value

Long. Returns the number of array elements written to *connectioninfo*.

Usage

A client application that has administrative privileges can use GetServerInfo to retrieve information about all clients connected to a particular server. In addition, once the connection information has been retrieved, a client with administrative privileges can use the RemoteStopConnection function to disconnect other client applications.

The server application grants connection privileges. The script for the server application's ConnectionBegin event can specify the privileges for each connection by returning a ConnectPrivilege value. A ConnectPrivilege value of ConnectWithAdminPrivilege! gives a client administrative privileges.

Examples

In this example, a client application uses GetServerInfo to retrieve information about all client connections established to the server specified in the myconnect Connection object. The connection information is loaded into a DataWindow:

```
connectioninfo info[]
int li_rownum
long ll_rows

ll_rows = myconnect.GetServerInfo(info)
dw_connections.reset()

for li_rownum = 1 to ll_rows
   dw_connections.insertrow(0)
```

```
        if info[li_rownum].busy then
           dw_connections.object.busy[li_rownum] = 1
        else
           dw_connections.object.busy[li_rownum] = 2
        end if
        dw_connections.object.connecttime[li_rownum] =
           info[li_rownum].connecttime
        dw_connections.object.lastcalltime[li_rownum]=
           info[li_rownum].lastcalltime
        dw_connections.object.callcount[li_rownum] =
           info[li_rownum].callcount
        dw_connections.object.clientid[li_rownum] =
           info[li_rownum].clientid
        dw_connections.object.connectstring[li_rownum] =
           info[li_rownum].connectstring
        dw_connections.object.connectuserid[li_rownum] =
           info[li_rownum].connectuserid
        dw_connections.object.location[li_rownum] =
           info[li_rownum].location
        dw_connections.object.userid[li_rownum] =
           info[li_rownum].userid
    next
```

See also RemoteStopConnection

GetShortName

Description

Gets the short name for the current PowerBuilder execution context.

Applies to

ContextInformation objects

Syntax

servicereference.**GetShortName** (*shortname*)

Argument	Description
servicereference	Reference to the ContextInformation service instance
shortname	String into which the function places the short name. This argument is passed by reference

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs. The function returns values as follows:

- **PowerBuilder execution-time** PBRun
- **PowerBuilder window plug-in** PBWinPlugin
- **PowerBuilder window ActiveX** PBRTX

Usage

Call this function to determine the current execution environment.

Examples

This example calls the GetShortName function. Ici_info is an instance variable of type ContextInformation:

```
String ls_name

this.GetContextService("ContextInformation", &
    ici_info)
ici_info.GetShortName(ls_name)
IF ls_name <> "PBRun" THEN
    cb_close.visible = FALSE
END IF
```

See also

GetCompanyName
GetContextService
GetFixesVersion
GetHostObject
GetMajorVersion
GetMinorVersion
GetName
GetVersionName

GetSpacing

Description

Obtains the line spacing of the paragraph containing the insertion point in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**GetSpacing** ( )

Argument	Description
rtename	The name of the RichTextEdit control in which you want to find out the line spacing of the paragraph containing the insertion point

Return value

Spacing. A value of the Spacing enumerated data type indicating the line spacing of the paragraph containing the insertion point.

Usage

When the user selects several paragraphs, the insertion point is at the beginning or end of the selection, depending on how the user made the selection. The value reported depends on the location of the insertion point.

Examples

This example stores a value of the enumerated data type spacing in the variable l_spacing. The value is the spacing for the paragraph with the insertion point:

```
spacing l_spacing
l_spacing = rte_1.GetSpacing()
```

See also

GetTextStyle
SetSpacing
SetTextStyle

GetSQLPreview

Description

Reports the SQL statement that the DataWindow control is currently submitting to the database.

Obsolete function
GetSQLPreview is obsolete and will be discontinued in the near future. You should replace all references to GetSQLPreview as soon as possible. The SQL syntax is available as an argument in the DBError and SQLPreview events.

Applies to

DataWindow controls and child DataWindows

Syntax

dwcontrol.**GetSQLPreview** ( )

Argument	Description
dwcontrol	The name of the DataWindow control or child DataWindow for which you want to obtain the SQL statement currently being submitted to the database server

Return value

String. Returns the current SQL statement for *dwcontrol*. Returns the empty string ("") if an error occurs. If *dwcontrol* is NULL, GetSQLPreview returns NULL.

Examples

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 (such as SQL Select, Quick Select, or Query).

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetSQLSelect** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow for which you want to obtain the current SELECT statement

Return value

String. Returns the current SQL SELECT statement for *dwcontrol*. GetSQLSelect returns the empty string ("") if it cannot return the statement. If *dwcontrol* is NULL, GetSQLSelect returns NULL.

Usage

When you want to change the SQL SELECT statement for a DataWindow or DataStore 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. The DataWindow object's Table.Select property holds the information.

FOR INFO For more information, see the *DataWindow Reference*.

Examples

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
Discussion of DataWindow object properties in the *DataWindow Reference*

GetStateStatus

Description

Retrieves the current status of the internal state flags for a DataWindow and places this information in a blob.

This function is used primarily in distributed applications.

Applies to

DataWindow controls and DataStore objects

Syntax

dwcontrol.**GetStateStatus** (*cookie*)

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore for which you want to get state status
cookie	A blob variable in which you want to store a cookie that contains state information for the DataWindow

Return value

Long. Returns 1 if it succeeds and -1 if it fails.

Usage

In situations where a single DataStore on a server acts as the source for multiple target DataWindows (or DataStores) on different clients, you can use GetChanges in conjunction with GetStateStatus to determine the likely success of SetChanges. This allows you to avoid shipping a change blob across the wire when the SetChanges call will fail anyway (because changes in the blob conflict with changes made previously by another client).

To determine the likely success of SetChanges, you need to:

1 Call the GetStateStatus function on the DataStore on which you want to do a SetChanges. GetStateStatus checks the state of the DataStore and makes the state information available in a reference argument called a **cookie**. The cookie is generally much smaller than a DataWindow change blob.

2 Send the cookie back to the client.

3 Call the GetChanges function on the DataWindow that contains the changes you want to apply, passing the cookie retrieved from GetStateStatus as a parameter. The return value from GetChanges indicates whether there are currently any potential conflicts between the state of the DataWindow blob and the state of the DataStore on which you want to execute SetChanges.

If the return value from GetChanges indicates that there are potential conflicts, you can then be certain that a subsequent call to SetChanges will fail if the FailOnAnyConflict! argument is specified. On the other hand, if the return value from GetChanges indicates no conflicts, the call to SetChanges may *still* fail, because the state of the Datastore may have changed since you called GetStateStatus and GetChanges. For example, if another client session has called SetChanges or some other processing has been executed that altered the state of the DataStore since you retrieved the cookie, then SetChanges will fail.

Examples

The following example is a script for a remote object function. The script uses GetStateStatus to capture the state of a DataStore on the server into a cookie. Once the cookie has been created, it is returned to the client:

```
blob lblb_cookie
long ll_rv

ll_rv = ids_datastore.GetStateStatus(lblb_cookie)

return lblb_cookie
```

See also

GetChanges
GetFullState
SetChanges
SetFullState

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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetText** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow for which you want to obtain the text of the current row and column

Return value

String. Returns the value in the edit control over the current row and column in *dwcontrol*. 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 *dwcontrol*. If *dwcontrol* is NULL, GetText returns NULL.

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.

You don't need to 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, use the data argument.

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.

Examples

These statements return the text in the edit control of dw_employee:

```
string LName
LName = dw_employee.GetText()
```

See also

SetText
GetText in the *DataWindow Reference*

GetTextColor

Description

Obtains the color of selected text in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**GetTextColor** ( )

Argument	Description
rtename	The name of the RichTextEdit control in which you want to find out the color of selected text

Return value

Long. Returns the long value that specifies the color of the currently selected text. If text of different colors is selected, GetTextColor returns the color of the first selected character. GetTextColor returns -1 if an error occurs.

Examples

This example stores a long representing the color of the selected text in rte_1:

```
long ll_color
ll_color = rte_1.GetTextColor()
```

See also

GetTextStyle
SetTextColor
SetTextStyle

GetTextStyle

Description

Finds out whether selected text has text styles (such as bold or italic) assigned to it.

Applies to

RichTextEdit controls

Syntax

rtename.**GetTextStyle** (*textstyle*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to find the formatting of selected text
textstyle	A value of the enumerated data type TextStyle specifying the text style you want to check for. Values are: Bold! Italic! Strikeout! Subscript! Superscript! Underlined!

Return value

Boolean. Returns TRUE if the selected text is formatted with the specified text style and FALSE if it is not. If *textstyle* is NULL, GetTextStyle returns NULL.

Usage

Text can be formatted with more than one text style. To test for different styles, call GetTextStyle more than once.

Examples

A previously defined structure is an instance variable istr_text for the current window. The structure contains the boolean fields: b_isBold, b_isItalic, and b_isUnderlined. This example checks whether the selected text has these styles and stores TRUE or FALSE values in the structure for each style:

```
istr_text.b_isBold = rte_fancy.GetTextStyle(Bold!)
istr_text.b_isItalic = rte_fancy.GetTextStyle(Italic!)
istr_text.b_isUnderlined = &
   rte_fancy.GetTextStyle(Underlined!)
```

See also

GetTextColor
SetSpacing
SetTextColor
SetTextStyle

GetToolbar

Description

Gets the current values for alignment, visibility, and title of the specified toolbar.

Applies to

MDI frame and sheet windows

Syntax

window.**GetToolbar** (*toolbarindex*, *visible* {, *alignment* {, *floatingtitle* } })

Argument	Description
window	The MDI frame or sheet to which the toolbar belongs
toolbarindex	An integer whose value is the index of the toolbar for which you want the current settings
visible	A boolean variable in which you want to store a value indicating whether the toolbar is visible
alignment (optional)	A variable of the ToolbarAlignment enumerated data type in which you want to store the current alignment of the toolbar
floatingtitle (optional)	A string variable in which you want to store the toolbar title that is displayed when the alignment is Floating!

Return value

Integer. Returns 1 if it succeeds. GetToolbar returns -1 if there is no toolbar for the index you specify or if an error occurs. If any argument's value is NULL, returns NULL.

Usage

To find out the position of the docked or floating toolbar, call GetToolbarPos.

Examples

This example finds out whether toolbar 1 is visible. It also gets the alignment and title of toolbar 1. The values are stored in the variables lb_visible, lta_align, and ls_title:

```
integer li_rtn
boolean lb_visible
toolbaralignment lta_align

li_rtn = w_frame.GetToolbar(1, lb_visible, &
   lta_align, ls_title)
```

This example displays the settings for the toolbar index the user specifies in sle_index. The IF and CHOOSE CASE statements convert the values to strings so they can be displayed in mle_toolbar:

```
integer li_index, li_rtn
boolean lb_visible
toolbaralignment lta_align
```

```
string ls_visible, ls_align, ls_title

li_index = Integer(sle_index.Text)
li_rtn = w_frame.GetToolbar(li_index, &
    lb_visible, lta_align, ls_title)

IF li_rtn = -1 THEN
    MessageBox("Toolbars", "Can't get toolbar
settings.")
    RETURN -1
END IF

IF lb_visible = TRUE THEN
    ls_visible = "TRUE"
ELSE
    ls_visible = "FALSE"
END IF

CHOOSE CASE lta_align
    CASE AlignAtTop!
        ls_align = "top"
    CASE AlignAtLeft!
        ls_align = "left"
    CASE AlignAtRight!
        ls_align = "right"
    CASE AlignAtBottom!
        ls_align = "bottom"
    CASE Floating!
        ls_align = "floating"
END CHOOSE

mle_1.Text = ls_visible + "~r~n" &
    + ls_align + "~r~n" &
    + ls_title
```

See also

GetToolbarPos
SetToolbar
SetToolbarPos

GetToolbarPos

Gets position information for the specified toolbar.

To get	Use
Docking position of a docked toolbar	Syntax 1
Coordinates and size of a floating toolbar	Syntax 2

Syntax 1 For docked toolbars

Description

Gets the position of a docked toolbar.

Applies to

MDI frame and sheet windows

Syntax

window.**GetToolbarPos** (*toolbarindex, dockrow, offset*)

Argument	Description
window	The MDI frame or sheet to which the toolbar belongs
toolbarindex	An integer whose value is the index of the toolbar for which you want the current settings
dockrow	An integer variable in which you want to store the number of the docking row for the specified toolbar. Docking rows are numbered from left to right or top to bottom
offset	An integer variable in which you want to store the offset of the toolbar from the beginning of the docking row. For toolbars at the top or bottom, *offset* is measured from the left edge. For toolbars at the left or right, *offset* is measured from the top

Return value

Integer. Returns 1 if it succeeds. GetToolbarPos returns -1 if there is no toolbar for the index you specify or if an error occurs. If any argument's value is NULL, returns NULL.

Usage

To find out whether the docked toolbar is at the top, bottom, left, or right edge of the window, call GetToolbar.

Syntax 1 for GetToolbarPos gets the most recent docked position, even if the toolbar is currently floating.

Examples

In this example, the user has specified a toolbar index in sle_2. The example gets the toolbar position information and displays it in a MultiLineEdit mle_1:

```
integer li_index, li_rtn
integer li_dockrow, li_offset

li_index = Integer(sle_2.Text)
li_rtn = w_frame.GetToolbarPos(li_index, &
   li_dockrow, li_offset)

// Report the position settings
IF li_rtn = 1 THEN
   mle_1.Text = String(li_dockrow) + "~r~n" &
      + String(li_offset)
ELSE
   mle_1.Text = "Can't get toolbar position"
END IF
```

See also

GetToolbar
SetToolbar
SetToolbarPos

Syntax 2

For floating toolbars

Description

Gets the position and size of a floating toolbar.

Applies to

MDI frame and sheet windows

Syntax

window.**GetToolbarPos** (*toolbarindex*, *x*, *y*, *width*, *height*)

Argument	Description
window	The MDI frame or sheet to which the toolbar belongs
toolbarindex	An integer whose value is the index of the toolbar for which you want the current settings
x	An integer variable in which you want to store the x coordinate of the floating toolbar. If the toolbar is docked, *x* is set to the most recent value
y	An integer variable in which you want to store the y coordinate of the floating toolbar. If the toolbar is docked, *y* is set to the most recent value
width	An integer variable in which you want to store the width of the floating toolbar. If the toolbar is docked, *width* is set to the most recent value

Argument	Description
height	An integer variable in which you want to store the height of the floating toolbar. If the toolbar is docked, *height* is set to the most recent value

Return value

Integer. Returns 1 if it succeeds. GetToolbarPos returns -1 if there is no toolbar for the index you specify or if an error occurs. If any argument's value is NULL, returns NULL.

Usage

To find out whether the toolbar is floating, call GetToolbar.

Syntax 2 for GetToolbarPos gets the most recent floating position, even if the toolbar is currently docked.

Examples

This example gets the x and y coordinates and the width and height of toolbar 1:

```
int ix, iy, iw, ih, li_rtn

li_rtn = w_frame.GetToolbarPos(1, ix, iy, iw, ih)
IF li_rtn = -1 THEN
   mle_1.Text = "Can't get toolbar position"
ELSE
   mle_1.Text = String(ix) + "~r~n" &
      + String(iy) + "~r~n" &
      + String(iw) + "~r~n" &
      + String(ih)
END IF
```

See also

GetToolbar
SetToolbar
SetToolbarPos

GetTrans

Description

Gets the values for the DataWindow control or DataStore object's internal transaction object and stores these values in the programmer-specified transaction object.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetTrans** (*transaction*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow for 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. If any argument's value is NULL, GetTrans returns NULL.

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.

Obsolete function
GetUpdateStatus is obsolete and will be discontinued in the near future. You should replace all references to GetUpdateStatus as soon as possible. The update status is available as an argument in the DBError and SQLPreview events.

Applies to

DataWindow controls and child DataWindows

Syntax

dwcontrol.**GetUpdateStatus** (*row*, *dwbuffer*)

Argument	Description
dwcontrol	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 (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*. If any argument's value is NULL, GetUpdateStatus returns NULL.

Examples

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
```

```
// Return 1 from the DBError event
// (do not display error message) because we've
// already displayed a message
RETURN 1
```

See also GetItemStatus

GetURL

Description

Returns HTML for the specified URL.

Applies to

Inet objects

Syntax

servicereference.**GetURL** (*urlname*, *data*)

Argument	Description
servicereference	Reference to the Internet service instance
urlname	String specifying the URL whose source data is returned in *data*
data	InternetResult descendant containing an overridden InternetData function that handles the HTML source for *urlname*

Return value

Integer. Returns values as follows:

1 Success
-1 General error
-2 Invalid URL
-4 Cannot connect to the Internet

Usage

Call this function to access HTML source for a URL.

Data references a standard class user object that descends from InternetResult and that has an overridden InternetData function. This overridden function then performs the processing you want with the returned HTML. Because the Internet returns data asynchronously, *data* must reference a variable that remains in scope after the function executes (such as a window-level instance variable).

FOR INFO For more information on the InternetResult standard class user object and the InternetData function, use the PowerBuilder Browser.

Examples

This example calls the GetURL function. Iinet_base is an instance variable of type inet:

```
iir_msgbox = CREATE n_ir_msgbox
iinet_base.GetURL(sle_url.text, iir_msgbox)
```

See also

HyperLinkToURL
InternetData
PostURL

GetValidate

Description

Obtains the validation rule for a column in a DataWindow.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetValidate** (*column*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol*. Returns the empty string ("") if no validation criteria are defined for the column. If any argument's value is NULL, GetValidate returns NULL.

Usage

You can use GetValidate to save the current validation rule before calling SetValidate to change the rule temporarily.

Examples

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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GetValue** (*column, index*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow in which you want to obtain the value of an item in a value list or 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 *dwcontrol*. 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. If any argument's value is NULL, GetValue returns NULL.

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 INFO 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

GetVersionName

Description

Gets complete version information for the current PowerBuilder execution context. A complete version includes a major version, a minor version, and a fix level (such as 6.0.03).

Applies to

ContextInformation objects

Syntax

servicereference.**GetVersionName** (*name*)

Argument	Description
servicereference	Reference to the ContextInformation service instance
name	String into which the function places the version name. This argument is passed by reference

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to determine the maintenance level of the current context.

Examples

This example calls the GetVersionName function. Ici_info is an instance variable of type ContextInformation:

```
String ls_name
String ls_version
Constant String ls_currver = "6.0.02"

GetContextService("ContextInformation", ici_info)
ici_info.GetVersionName(ls_version)
IF ls_version <> ls_currver THEN
   MessageBox("Error", &
      "Must be at Version " + ls_currver)
END IF
```

See also

GetCompanyName
GetFixesVersion
GetHostObject
GetMajorVersion
GetMinorVersion
GetName
GetShortName

GroupCalc

Description

Recalculates the breaks in the grouping levels in a DataWindow.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**GroupCalc** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If *dwcontrol* is NULL, GroupCalc returns NULL.

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.

Examples

This code imports new rows from a file into the DataWindow dw_emp and then recalculates the group breaks for dw_emp:

```
dw_emp.ImportFile("d:\employee.txt")
dw_emp.SetRedraw(false)
dw_emp.SetSort("1A")
dw_emp.Sort()
dw_emp.GroupCalc()
dw_emp.SetRedraw(true)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
dw_emp.ImportFile("HD:employee.txt")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
dw_emp.ImportFile("/export/home/employee.txt")
```

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* })

Argument	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 the PowerBuilder 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 the PowerBuilder 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. On Windows 3.1, you can use Handle with this argument to prevent the user from running multiple instances of your application (see the examples). On Windows 95 and Windows NT, the Handle function does not return a useful value when the previous flag is TRUE.

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. You can use this technique if you are deploying your application on Windows 3.1:

```
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
```

On Windows 95 and Windows NT, the Handle function does not return a useful value when the previous flag is TRUE. You can use the FindWindowA Windows function to determine whether a Windows application is already running.

Declare FindWindowA as a global external function:

```
FUNCTION uint FindWindowA (long classname,  &
   string windowname) LIBRARY "user32.dll"
```

Then add code like the following to your application's open event:

```
uint  val
val = FindWindowA(0, "MyApp Main Window")
IF val > 0 THEN
   MessageBox("Application already running", &
   "MyApp 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** ( )

Argument	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. If *objectname* is NULL, Hide returns NULL.

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 property.

Equivalent syntax You can set an object's Visible property 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*)

Argument	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*. If *time* is NULL, Hour returns NULL.

Examples

This statement returns the current hour:

```
Hour(Now())
```

This statement returns 19:

```
Hour(19:01:31)
```

See also

Minute
Now
Second
Hour in the *DataWindow Reference*

HyperLinkToURL

Description

Opens the default Web browser, displaying the specified URL.

Applies to

Inet objects

Syntax

servicereference.**HyperlinkToURL** (*url*)

Argument	Description
servicereference	Reference to the Internet service instance
url	String specifying the URL to open in the default Web browser

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to display a URL from a PowerBuilder application.

Examples

This example calls the HyperlinkToURL function. Iinet_base is an instance variable of type inet:

```
GetContextService("Internet", iinet_base)
iinet_base.HyperlinkToURL(sle_url.text)
```

See also

GetURL
PostURL

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*)

Argument	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. If *n* is NULL, Idle returns NULL. 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 (such as, 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)
- Any retrieval on a visible DataWindow that causes the edit control to be painted

Tip

To capture movement, write script in the MouseMove or Key events of the window or sheet. (Keyboard activity does not trigger MouseMove events.) Disable the DataWindow control and tab ordering during iterative retrieves so the Idle timer is not reset.

Examples

This statement sends an Idle event after five minutes of inactivity:

```
Idle(300)
```

This statement turns off idle detection:

```
Idle(0)
```

This 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 this statement 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

Inserts data into a DataWindow control, DataStore object, or graph control from tab-delimited data on the clipboard.

To	Use
Import rows into a DataWindow control or DataStore object	Syntax 1
Add new series to a graph control	Syntax 2

Syntax 1

For DataWindows and DataStores

Description

Inserts data into a DataWindow control or DataStore object from tab-delimited data on the clipboard.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**ImportClipboard** ({ *startrow* {, *endrow* {, *startcolumn* {, *endcolumn* {, *dwstartcolumn* } } } } })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 or DataStore 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

If any argument's value is NULL, ImportClipboard returns NULL.

Usage

The clipboard data must be formatted in tab-delimited columns. The data types and order of the DataWindow object'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 column to be affected.

dwstartcolumn + (*endcolumn* - *startcolumn*)

Examples

This statement copies all data in the clipboard to the DataWindow dw_employee starting at the first column:

```
dw_employee.ImportClipboard()
```

This 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. It adds 29 rows to the DataWindow with data in columns 5 through 10:

```
dw_employee.ImportClipboard(2,30,3,8,5)
```

See also

ImportFile
ImportString

Syntax 2 For graph controls

Description

Inserts data into a graph control from tab-delimited data on the clipboard.

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

graphname.**ImportClipboard** ({ *startrow* {, *endrow* {, *startcolumn* } } })

Argument	Description
graphname	The name of the graph control to which you want to copy data from the clipboard

Argument	Description
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

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

If any argument's value is NULL, ImportClipboard returns NULL.

Usage

The clipboard data must be formatted in tab-delimited columns.

For graphs, 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

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 94Jan3000
Sales 94Mar2200
Sales 94May2500
Sales 95Jan4000
```

```
Sales 95Mar3200
Sales 95May3500
```

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

Inserts data into a DataWindow control, DataStore object, 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 DataStore) or a graph and on the type of graph.

To	Use
Import rows into a DataWindow control or DataStore	Syntax 1
Add new series to a graph control	Syntax 2

Syntax 1

For DataWindows and DataStores

Description

Inserts data into a DataWindow control or DataStore from a file. The data can be tab-delimited text or dBase format 2 or 3.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**ImportFile** (*filename* {, *startrow* {, *endrow* {, *startcolumn* {, *endcolumn* {, *dwstartcolumn* } } } } })

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore 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. If *filename* is NULL, ImportFile displays the File Open dialog box and allows the user to select a file
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

Argument	Description
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 or DataStore that you want to receive data. The default is 1

Events

ImportFile may trigger an ItemError event.

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:

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
-10 Unsupported dBase file format (not version 2 or 3)

Usage

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 object'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*)

To let users select the file to import, specify a NULL string for *filename*. PowerBuilder displays the Select Import File dialog box. A dropdown listbox lets the user select the type of file to import.

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")
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
dw_employee.ImportFile("HD:Employee")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
dw_employee.ImportFile("/export/home/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)
```

See also

ImportClipboard
ImportString

Syntax 2

For graph controls

Description

Inserts data into a 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 the type of 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

graphname.**ImportFile** (*filename* {, *startrow* {, *endrow* {, *startcolumn* } } })

Argument	Description
graphname	The name of the graph control to which you want to copy data from the specified file
filename	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* or if it is NULL, ImportFile prompts the user for a filename

Argument	Description
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

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

Usage

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 copies all the data in the file D:\EMPLOYEE.TXT to gr_employee starting at the first row:

```
gr_employee.ImportFile("D:\EMPLOYEE.TXT")
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
gr_employee.ImportFile("HD:Employee")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
gr_employee.ImportFile("/export/home/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)
```

This 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

Inserts data into a DataWindow control, DataStore object, or graph control from tab-delimited data in a string. The way data is arranged in the string in tab-delimited columns depends on whether the target is a DataWindow (or DataStore) or a graph, and on the type of graph.

To	Use
Import rows into a DataWindow control or DataStore	Syntax 1
Add new series to a graph control	Syntax 2

Syntax 1 For DataWindows and DataStores

Description

Inserts data into a DataWindow control or DataStore from tab-delimited data in a string.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**ImportString** (*string* {, *startrow* {, *endrow* {, *startcolumn* {, *endcolumn* {, *dwstartcolumn* } } } } })

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore 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 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 all 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

Argument	Description
dwstartcolumn (optional)	The number of the first column in the DataWindow control or DataStore object that you want to receive data. The default is 1

Events

ImportString may trigger an ItemError event.

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:

- -1 *startrow* value supplied is greater than the number of rows in the string
- -3 Invalid argument
- -4 Invalid input
- -9 PowerBuilder or the user canceled import because data failed validation

If any argument's value is NULL, ImportString returns NULL.

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~t *col2_data*~t *col3_data*~t *col4_data*~r~n

For a DataWindow control or DataStore, 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 object'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*)

If string data to be assigned to a single row and column has multiple lines (indicated by line ending characters in the import string), you must quote the string data using ~". Do not use single quotes.

This example of a valid import string assigns multiline values to each row in column 2:

```
ls_s = &
    "1~t~"Mickey~r~nMinnie~r~nGoofy~" ~r~n" + &
```

```
"2~t~"Susan~r~nMary~r~nMarie~" ~r~n" + &
"3~t~"Chris~r~nBen~r~nMike~" ~r~n" + &
"4~t~"Mott~r~nBarber~r~nPicard~" "
```

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)
```

This 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)
```

See also

ImportClipboard
ImportFile

Syntax 2 or graph controls

Description

Inserts data into a graph control from tab-delimited data in a string. The format of the String depends on the type of 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

graphname.**ImportString** (*string* {, *startrow* {, *endrow* {, *startcolumn* } } })

Argument	Description
graphname	The name of the graph 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'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

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
-1 *startrow* value supplied is greater than the number of rows in the string
-2 Not enough columns
-3 Invalid argument
-4 Invalid input

If any argument's value is NULL, ImportString returns NULL.

Usage

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 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

IncomingCallList

Description

Provides a list of the callers of a routine included in a performance analysis model.

Applies to

ProfileRoutine object

Syntax

iinstancename.**IncomingCallList** (*list*, *aggregrateduplicateroutinecalls*)

Argument	Description
instancename	Instance name of the ProfileRoutine object
list	An unbounded array variable of data type ProfileCall in which IncomingCallList stores a ProfileCall object for each caller of the routine. This argument is passed by reference
aggregateduplicateroutinecalls	A boolean indicating whether duplicate routine calls will result in the creation of a single or of multiple ProfileCall objects

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- ModelNotExistsError!—The model does not exist

Usage

Use this function to extract a list of the callers of a routine included in a performance analysis model. Each caller is defined as a ProfileCall object and provides the called routine and the calling routine, the number of times the call was made, and the elapsed time. The callers are listed in no particular order.

You must have previously created the performance analysis model from a trace file using the BuildModel function.

The *aggregateduplicateroutinecalls* argument indicates whether duplicate routine calls will result in the creation of a single or of multiple ProfileCall objects. This argument has no effect unless line tracing is enabled and a calling routine calls the current routine from more than one line. If *aggregateduplicateroutinecalls* is TRUE, a new ProfileCall object is created that aggregates all calls from the calling routine to the current routine. If *aggregateduplicateroutinecalls* is FALSE, multiple ProfileCall objects are returned, one for each line from which the calling routine called the called routine.

Examples

This example gets a list of the routines included in a performance anaylsis model and then gets a list of the routines that called each routine:

```
Long ll_cnt
ProfileCall lproc_call[]

lpro_model.BuildModel()
lpro_model.RoutineList(i_routinelist)

FOR ll_cnt = 1 TO UpperBound(iprort_list)
   iprort_list[ll_cnt].IncomingCallList(lproc_call, &
   TRUE)
   ...
NEXT
```

See also

BuildModel
OutgoingCallList

InputFieldChangeData

Description

Modifies the data value of input fields in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**InputFieldChangeData** (*inputfieldname*, *inputfieldvalue*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to change the data in the specified input fields
inputfieldname	A string whose value is the name of input fields whose value you want to change. There can be more than one input field with a given name
inputfieldvalue	A string whose value is the data to be assigned to the specified input fields

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, InputFieldChangeData returns NULL.

Usage

All the input fields that have the same name contain the same data. When you call InputFieldChangeData, you affect all the fields of the specified name.

Examples

This script is part of the SelectionChanged event for the ListBox lb_instruments. When the user clicks on an item in the ListBox, the selected instrument name is assigned to the input field called instrument in the RichTextEdit rte_1:

```
integer rtn
rtn = rte_1.InputFieldChangeData("instrument",
lb_instruments.SelectedItem())

st_status.Text = String(rtn)
```

If the text in rte_1 looks like this:

Dear {title} {lastname}:

We're happy you have rented a {instrument} for your child. Please perform regular maintenance for the {instrument} as instructed by your child's teacher. You can buy {instrument} supplies and instruction books at your local music stores.

Then after the user picks *trumpet* in the ListBox, the script inserts *trumpet* for every occurrence of the {instrument} field. The other fields are not affected:

Dear {title} {lastname}:

We're happy you have rented a trumpet for your child. Please perform regular maintenance for the trumpet as instructed by your child's teacher. You can buy trumpet supplies and instruction books at your local music stores.

See also

InputFieldCurrentName
InputFieldDeleteCurrent
InputFieldGetData
InputFieldInsert
InputFieldLocate
DataSource

InputFieldCurrentName

Description

Gets the name of the input field when the insertion point is in an input field in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**InputFieldCurrentName** ( )

Argument	Description
rtename	The name of the RichTextEdit control in which you want to get the input field's name

Return value

String. Returns the name of the input field. If the insertion point is not in an input field or if an error occurs, it returns the empty string ("").

Examples

This example gets the name of the input field containing the insertion point:

```
string ls_inputname
ls_inputname = rte_1.InputFieldCurrentName()
```

See also

InputFieldChangeData
InputFieldDeleteCurrent
InputFieldGetData
InputFieldInsert
InputFieldLocate
DataSource

InputFieldDeleteCurrent

Description

Deletes the input field that is selected in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**InputFieldDeleteCurrent** ( )

Argument	Description
rtename	The name of the RichTextEdit control in which you want to delete the input field that is selected

Return value

Integer. Returns 1 if it succeeds and -1 if there is no input field at the insertion point, the input field is activated for editing, or an error occurs.

Usage

All the input fields that have the same name contain the same data but they can be deleted independently. If one of a group of input fields with the same name is deleted, the others are not affected. If all the input fields of the same name are deleted, the RichTextEdit control remembers the data from those input fields. It will use that data to initialize a new input field that has the same name as the deleted fields.

The input field must be the only selection. If other text is selected too, InputFieldDeleteCurrent fails. When an input field is the current and only selection, the highlight flashes.

InputFieldDeleteCurrent deletes only the current field. Other fields with the same name within the document are not affected. If the RichTextEdit control uses the DataSource function to share data with a DataWindow, the current field is deleted from all instances of the document.

Examples

This example deletes the input field containing the insertion point:

```
integer li_rtn
li_rtn = rte_1.InputFieldDeleteCurrent()
```

See also

InputFieldChangeData
InputFieldGetData
InputFieldCurrentName
InputFieldInsert
InputFieldLocate
DataSource

InputFieldGetData

Description

Get the data in the specified input field in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**InputFieldGetData** (*inputfieldname*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to get data from the selected input field
inputfieldname	A string whose value is the name of input field from which you want to get the data

Return value

String. The data in the input field. InputFieldGetData returns the empty string ("") if the field doesn't exist or an error occurs.

Examples

This example gets the data in the input field empname:

```
string ls_name
ls_name = rte_1.InputFieldGetData(empname)
```

See also

InputFieldChangeData
InputFieldCurrentName
InputFieldDeleteCurrent
InputFieldInsert
InputFieldLocate
DataSource

InputFieldInsert

Description

Inserts a named input field at the insertion point in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**InputFieldInsert** (*inputfieldname*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to insert an input field
inputfieldname	A string whose value is the name of input field to be inserted. The name does not have to be unique

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *inputfieldname* is NULL, InputFieldInsert returns NULL.

Usage

There can be several input fields with the same name. Fields of a given name all have the same data value. When you call InputFieldChangeData for a named input field, all fields with that name are changed.

If there is a selection, InputFieldInsert inserts the field at the beginning of the selection. The input field and the selection remain selected:

Examples

```
st_status.Text = String( &
rte_1.InputFieldInsert("lastname"))
```

See also

InputFieldChangeData
InputFieldCurrentName
InputFieldDeleteCurrent
InputFieldGetData
InputFieldLocate
DataSource

InputFieldLocate

Description

Locates an input field in a RichTextEdit control and moves the insertion point there.

Applies to

RichTextEdit controls

Syntax

rtename.**InputFieldLocate** (*location* {, *inputfieldname* })

Argument	Description
rtename	The name of the RichTextEdit control in which you want to locate an input field
location	A value of the Location enumerated data type specify the occurrence of the input field you want to locate. Values are: ♦ First! — The first occurrence in the document of *inputfieldname*, or if no name is specified, the first input field in the document ♦ Last! — The last occurrence in the document of *inputfieldname*, or if no name is specified, the last input field in the document ♦ Next! — The occurrence of *inputfieldname* that is after the insertion point, or if no name is specified, the next input field of any name after the insertion point ♦ Prior! — The occurrence of *inputfieldname* before the insertion point, or if no name is specified, the next input field of any name before the insertion point
inputfieldname	A string whose value is the name of input field you want to locate. If there are multiple occurrences of *inputfieldname* in the control, *location* specifies the one to be located

Return value

String. Returns the name of the input field it located if it succeeds. InputFieldLocate returns an empty string if no matching input field is found or if an error occurs. If any argument is NULL, InputFieldLocate returns NULL.

Usage

There can be several input fields with the same name. Fields of a given name all have the same data value.

Examples

This example locates the next input field after the insertion point. If found, ls_name is set to the name of the input field:

```
string ls_name
```

```
ls_name = rte_1.InputFieldLocate(Next!)
```

This example locates the last input field in the document:

```
string ls_name
ls_name = rte_1.InputFieldLocate(Last!)
```

This example locates the last occurrence in the document of the input field named address. If found, ls_name is set to the value "address":

```
string ls_name
ls_name = rte_1.InputFieldLocate(Last!, "address")
```

See also

InputFieldChangeData
InputFieldCurrentName
InputFieldDeleteCurrent
InputFieldGetData
InputFieldInsert
DataSource

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*)

Argument	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
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. If any argument's value is NULL, InsertCategory returns NULL.

Usage

Categories are discrete. 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")
```

Examples

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. If the axis is sorted, the order in which you add the categories doesn't matter:

```
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 control.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Syntax

ole2control.**InsertClass** (*classname*)

Argument	Description
ole2control	The name of the OLE 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

If any argument's value is NULL, InsertClass returns NULL.

Usage

Classnames are stored in the Registration database. Examples of classnames include:

Excel.Sheet
Excel.Chart
Word.Document

Examples

This example inserts an empty Excel spreadsheet into the OLE control, ole_1:

```
integer result
result = ole_1.InsertClass("excel.sheet")
```

See also

InsertFile
InsertObject
LinkTo

InsertColumn

Description

Inserts a column with the specified label, alignment, and width at the specified location.

Applies to

ListView controls

Syntax

listviewname.**InsertColumn** (*index, label, alignment, width*)

Argument	Description
listviewname	The name of the ListView control to which you want to insert a column
index	An integer whose value is the number of the column before which you are inserting a new column
label	A string whose value is the name of the column you are inserting
alignment	A value of the enumerated data type Alignment specifying the alignment of the column you are inserting. Values are: ♦ Center! ♦ Justify! ♦ Left! ♦ Right!
width	An integer whose value is the width of the column you are inserting

Return value

Integer. Returns the column *index* value if it succeeds and -1 if an error occurs.

Usage

Columns can not be inserted before the first column.

Examples

This example inserts a column named Location, makes it right-aligned, and sets the column width to 300:

```
lv_list.InsertColumn(2 , "Location" , Right! , 300)
```

See also

AddColumn
DeleteColumn

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* })

Argument	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. If any argument's value is NULL, InsertData returns NULL.

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

InsertDocument

Description

Inserts a rich text format or plain text file into a RichTextEdit control, DataWindow control, or DataStore object. The new content is added in one of two ways:

- The new content can be inserted at the insertion point.
- The new content can replace all existing content.

Applies to

RichTextEdit controls, DataWindow controls, and DataStore objects

Syntax

rtename.**InsertDocument** (*filename, clearflag* { , *filetype* })

Argument	Description
rtename	The name of the RichTextEdit control, DataWindow control, or DataStore object in which you want to display the file. The DataWindow object in the DataWindow control (or DataStore) must be a RichTextEdit DataWindow
filename	A string whose value is the name of the file you want to display in the RichTextEdit control. *Filename* can include the file's path
clearflag	A boolean value specifying whether the new file will replace the current contents of the control. Values are: ♦ True — Replace the current contents with the file ♦ False — Insert the file into the existing contents at the insertion point
filetype (optional)	A value of the FileType enumerated data type specifying the type of file being opened. Values are: ♦ FileTypeRichText! — (Default) The file being opened is in rich text format (RTF) ♦ FileTypeText! — The file being opened is plain ASCII text (TXT) If *filetype* is not specified, PowerBuilder uses the filename's extension to decide whether to read the file as rich text or plain text. If the extension is not RTF or TXT, PowerBuilder reads the file as plain text

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, InsertDocument returns NULL.

Usage

When the control supports headers and footer (the HeaderFooter property is set to TRUE), inserting a document affects existing header and footer text. If *clearflag* is set to FALSE, header and footer text from the inserted document is added to existing header and footer text.

Not all RTF formatting is supported. PowerBuilder supports version 1.2 of the RTF standard, except for the following:

- No support for formatted tables
- No drawing objects
- No double-underline

Any unsupported formatting is ignored.

Examples

This example inserts a document into rte_1 and reports the return value in a StaticText control:

```
integer rtn
rtn = rte_1.InsertDocument("c:\pb\test.rtf", &
TRUE, FileTypeRichText!)
st_status.Text = String(rtn)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
rtn = rte_1.InsertDocument("HD:PB:Test Rich Text", &
TRUE, FileTypeRichText!)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
rtn = rte_1.InsertDocument( &
"/export/home/pb/test.rtf", TRUE, &
FileTypeRichText!)
```

See also

InputFieldInsert
InsertPicture
DataSource

InsertFile

Description

Inserts an object into an OLE control. A copy of the specified file is embedded in the OLE object.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Syntax

olecontrol.**InsertFile** (*filename*)

Argument	Description
olecontrol	The name of the OLE 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

If any argument's value is NULL, InsertFile returns NULL.

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.

Examples

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

Inserts an item into a ListBox, DropDownListBox, ListView or TreeView control.

To insert an item into a	Use
ListBox or DropDownListBox control	Syntax 1
PictureListBox or DropDownPictureListBox control	Syntax 2
ListView control when only the label and picture index need to be specified	Syntax 3
ListView control when more than the label and picture index need to be specified	Syntax 4
TreeView control when only the label and picture index need to be specified	Syntax 5
TreeView control when more than the label and picture index need to be specified	Syntax 6

Syntax 1

For ListBox and DropDownListBox controls

Description

Inserts an item into the list of values in a listbox.

Applies to

ListBox and DropDownListBox controls

Syntax

listboxname.**InsertItem** (*item*, *index*)

Argument	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. If any argument's value is NULL, InsertItem returns NULL.

Usage

InsertItem inserts the new item before the item identified by *index*. If the items in *listboxname* are sorted (its Sorted property 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.

AddItem and InsertItem do not update the Items property array. You can use FindItem to find items added during execution.

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 property is FALSE, the statement above returns 5 (the previous item 5 becomes item 6). If the Sorted property 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 property 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 property 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

Syntax 2

For ListBox and DropDownListBox controls

Description

Inserts an item into the list of values in a picture listbox.

Applies to

PictureListBox and DropDownPictureListBox controls

Syntax

listboxname.**InsertItem** (*item* {, *pictureindex* }, *index*)

Argument	Description
listboxname	The name of the PictureListBox or DropDownPictureListBox into which you want to insert an item
item	A string whose value is the text of the item you want to insert
pictureindex (optional)	An integer specifying the index of the picture you want to associate with the newly added item
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. If any argument's value is NULL, InsertItem returns NULL.

Usage

If you don't specify a picture index, the newly added item will not have a picture.

If you specify a picture index that doesn't exist, that number is still stored with the picture. If you add pictures to the picture array so that the index becomes valid, the item will then show the corresponding picture.

FOR INFO For additional notes about items in ListBoxes and examples of how the Sorted property affects the item order, see Syntax 1.

Examples

This statement inserts the item Run Application before the fifth item in lb_actions. The item has no picture assigned to it:

```
plb_actions.InsertItem("Run Application", 5)
```

This statement inserts the item Run Application before the fifth item in lb_actions and assigns it picture index 4:

```
plb_actions. nsertItem("Run Application", 4, 5)
```

See also

AddItem
DeleteItem
FindItem
Reset
TotalItems

Syntax 3 For ListView controls

Description

Inserts an item into a ListView control.

Applies to

ListView controls

Syntax

listviewname.**InsertItem** (*index, label, pictureindex*)

Argument	Description
listviewname	The name of the ListView control to which you are adding an item
index	An integer whose value is the index number of the item before which you are inserting a new item
label	A string whose value is the name of the item you are adding
pictureindex	An integer whose value is the index number of the picture of the item you are adding

Return value

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

Usage

If you need to set more than the label and picture index, use Syntax 4.

Examples

This example inserts an item in the ListView in position 11:

```
lv_list.InsertItem(11 , "Presentation" , 1)
```

See also

AddItem

Syntax 4 For ListView controls

Description

Inserts an item into a ListView control.

Applies to

ListView controls

Syntax

listviewname.**InsertItem** (*index, item*)

Argument	Description
listviewname	The name of the ListView control into which you are inserting an item
index	An integer whose value is the index number of the item you are adding

Argument	Description
item	A system structure of data type ListViewItem in which InsertItem stores the item you are inserting

Return value

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

Usage

The index you specify is the position of the item you are adding to a ListView.

If you need to insert just the label and picture index into the ListView control, use Syntax 3.

Examples

This example moves a ListView item from the second position into the fifth position. It uses GetItem to retrieve the state information from item 2, inserts it into the ListView control as item 5, and then deletes the original item:

```
listviewitem l_lvi

lv_list.GetItem(2 , l_lvi)
lv_list.InsertItem(5 , l_lvi)
lv_list.DeleteItem(2)
```

See also

AddItem

Syntax 5

For TreeView controls

Description

Inserts an item at a specific level and order in a TreeView control.

Applies to

TreeView controls

Syntax

treeviewname.**InsertItem** (*handleparent, handleafter, label, pictureindex*)

Argument	Description
treeviewname	The name of the TreeView control in which you want to insert an item
handleparent	The handle of the item one level above the item you want to insert. To insert an item at the first level, specify 0
handleafter	The handle of the item on the same level that you will insert the item immediately after
label	The label of the item you are inserting
pictureindex	The Index of the index of the picture you are adding to the image list

Return value

Long. Returns the handle of the inserted item if it succeeds and -1 if an error occurs.

Usage

Use this syntax to set just the label and picture index. Use the next syntax if you need to set additional properties for the item.

If the TreeView's SortType property is set to a value other than Unsorted!, the inserted item is sorted with its siblings.

If you are inserting the first child of an item, use InsertItemLast or InsertItemFirst instead. Those functions don't require a *handleafter* value.

Examples

This example inserts a TreeView item that is on the same level as the current TreeView item. It uses FindItem to get the current item and its parent, then inserts the new item beneath the parent item:

```
long ll_tvi, ll_tvparent
ll_tvi = tv_list.FindItem(currenttreeitem! , 0)
ll_tvparent = tv_list.FindItem(parenttreeitem!,ll_tvi)
tv_list.InsertItem(ll_tvparent,ll_tvi,"Hindemith", 2)
```

See also

GetItem

Syntax 6

For TreeView controls

Description

Inserts an item at a specific level and order in a TreeView control.

Applies to

TreeView controls

Syntax

treeviewname.**InsertItem** (*handleparent, handleafter, item*)

Argument	Description
treeviewname	The name of the TreeView control into which you want to insert an item
handleparent	The handle of the item one level above the item you want to insert. To insert an item at the first level, specify 0
handleafter	The handle of the item on the same level that you will insert the item immediately after
item	A TreeViewItem structure for the item you are inserting

Return value

Long. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.

Usage

Use the previous syntax to set just the label and picture index. Use this syntax if you need to set additional properties for the item.

If the TreeView's SortType property is set to a value other than Unsorted!, the inserted item is sorted with its siblings.

If you are inserting the first child of an item, use InsertItemLast or InsertItemFirst instead. Those functions don't require a *handleafter* value.

Examples

This example inserts a TreeView item that is on the same level as the current TreeView item. It uses FindItem to get the current item and its parent, then inserts the new item beneath the parent item:

```
long ll_tvi, ll_tvparent
ll_tvi = tv_list.FindItem(currenttreeitem!, 0)
ll_tvparent = tv_list.FindItem(parenttreeitem!,ll_tvi)
tv_list.InsertItem(ll_tvparent,ll_tvi,"Hindemith")
```

See also

GetItem

InsertItemFirst

Inserts an item as the first child of a parent item.

To insert an item as the first child of its parent	Use
When you only need to specify the item label and picture index	Syntax 1
When you need to specify more than the item label and picture index	Syntax 2

Syntax 1 For TreeView controls

Description

Inserts an item as the first child of its parent.

Applies to

TreeView controls

Syntax

treeviewname.**InsertItemFirst** (*handleparent, label, pictureindex*)

Argument	Description
treeviewname	The TreeView control in which you want to specify an item as the first child of its parent
handleparent	The handle of the item that will be the inserted item's parent. To insert the item at the first level, specify 0
label	The label of the item you want to specify as the first child of its parent
pictureindex	The picture index for the item you want to specify as the first child of its parent

Return value

Long. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.

Usage

Examples

This example populates the first level of a TreeView using InsertItemFirst:

```
long ll_lev1, ll_lev2 ,ll_lev3 ,ll_lev4
int index

tv_list.PictureHeight = 32
tv_list.PictureWidth = 32

ll_lev1 = tv_list.InsertItemFirst(0,"Composers",1)
```

```
ll_lev2 = tv_list.InsertItemLast(ll_lev1,&
   "Beethoven",2)
ll_lev3 = tv_list.InsertItemLast(ll_lev2,&
   "Symphonies", 3)
FOR index = 1 to 9
   ll_lev4 = tv_list.InsertItemSort(ll_lev3, &
      "Symphony # " + String(index) , 4)
NEXT

tv_list.ExpandItem(ll_lev3)
tv_list.ExpandItem(ll_lev4)
```

See also

InsertItem
InsertItemLast
InsertItemSort

Syntax 2

For TreeView controls

Description

Inserts an item as the first child of an item.

Applies to

TreeView controls

Syntax

treeviewname.**InsertItemFirst** (*handleparent, item*)

Argument	Description
treeviewname	The TreeView control in which you want to specify an item as the first child of its parent
handleparent	The handle of the item that will be the inserted item's parent. To insert the item at the first level, specify 0
item	A TreeViewItem structure for the item you are inserting

Return value

Long. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.

Usage

If SortType is anything except Unsorted!, items are sorted after they are added and the TreeView is always in a sorted state. Therefore, calling InsertItemFirst, InsertItemLast, and InsertItemSort produces the same result.

Examples

This example inserts the current item as the first item beneath the root item in a TreeView control:

```
long          ll_handle, ll_roothandle
```

```
treeviewitem  l_tvi
ll_handle = tv_list.FindItem(CurrentTreeItem!, 0)
ll_roothandle = tv_list.FindItem(RootTreeItem!, 0)
tv_list.GetItem(ll_handle , l_tvi)

tv_list.InsertItemFirst(ll_roothandle, l_tvi)
```

See also

InsertItem
InsertItemLast
InsertItemSort

InsertItemLast

Inserts an item as the last child of a parent item.

To insert an item as the last child of its parent	Use
When you only need to specify the item label and picture index	Syntax 1
When you need to specify more than item label and picture index	Syntax 2

Syntax 1

For TreeView controls

Description

Inserts an item as the last child of its parent.

Applies to

TreeView controls

Syntax

treeviewname.**InsertItemLast** (*handleparent, label, pictureindex*)

Argument	Description
treeviewname	The TreeView control in which you want to specify an item as the last child of its parent
handleparent	The handle of the item that will be the inserted item's parent. To insert the item at the first level, specify 0
label	The label of the item you want to specify as the last child of its parent
pictureindex	The picture index for the item you want to specify as the last child of its parent

Return value

Long. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.

Usage

If more than the item label and Index need to be specified, use syntax 2.

If SortType is anything except Unsorted!, items are sorted after they are added and the TreeView is always in a sorted state. Therefore, calling InsertItemFirst, InsertItemLast, and InsertItemSort produces the same result.

Examples

This example populates the first three levels of a TreeView using InsertItemLast:

```
long  ll_lev1, ll_lev2, ll_lev3, ll_lev4
```

```
int   index

tv_list.PictureHeight = 32
tv_list.PictureWidth = 32

ll_lev1 = tv_list.InsertItemLast(0,"Composers",1)
ll_lev2 = tv_list.InsertItemLast(ll_lev1,&
   "Beethoven",2)
ll_lev3 = tv_list.InsertItemLast(ll_lev2,&
   "Symphonies",3)
FOR index = 1 to 9
   ll_lev4 = tv_list.InsertItemSort(ll_lev3, &
      "Symphony # " String(index), 4)
NEXT

tv_list.ExpandItem(ll_lev3)
tv_list.ExpandItem(ll_lev4)
```

See also

InsertItem
InsertItemFirst
InsertItemSort

Syntax 2 For TreeView controls

Description

Inserts an item as the last child of its parent.

Applies to

TreeView controls

Syntax

treeviewname.**InsertItemLast** (*handleparent, item*)

Argument	Description
treeviewname	The TreeView control in which you want to specify an item as the last child of its parent
handleparent	The handle of the item that will be the inserted item's parent. To insert the item at the first level, specify 0
item	A TreeViewItem structure for the item you are inserting

Return value

Long. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.

Usage

If SortType is anything except Unsorted!, items are sorted after they are added and the TreeView is always in a sorted state. Therefore, calling InsertItemFirst, InsertItemLast, and InsertItemSort produces the same result.

Examples

This example inserts the current item as the last item beneath the root item in a TreeView control:

```
long         ll_handle, ll_roothandle
treeviewitem l_tvi

ll_handle = tv_list.FindItem(CurrentTreeItem!, 0)
ll_roothandle = tv_list.FindItem(RootTreeItem!, 0)
tv_list.GetItem(ll_handle , l_tvi)

tv_list.InsertItemLast(ll_roothandle, l_tvi)
```

See also

InsertItem
InsertItemFirst
InsertItemSort

InsertItemSort

Inserts a child item in sorted order under the parent item.

To insert an item in sorted order	Use
When you only need to specify the item label and picture index	Syntax 1
When you need to specify more than the item label and picture index	Syntax 2

Syntax 1

For TreeView controls

Description

Inserts an item in sorted order, if possible.

Applies to

TreeView controls

Syntax

treeviewname.**InsertItemSort** (*handleparent, label, pictureindex*)

Argument	Description
treeviewname	The TreeView control in which you want to insert and sort an item as a child of its parent, according to its label
handleparent	The handle of the item that will be the inserted item's parent. To insert the item at the first level, specify 0
label	The label by which you want to sort the item as a child of its parent
pictureindex	The picture index for the item you want to sort as a child of its parent, according to its label

Return value

Long. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.

Usage

If SortType is anything except Unsorted!, the TreeView is always in a sorted state and you don't need to use InsertItemSort—you can use any insert function.

If SortType is Unsorted!, InsertItemSort attempts to insert the item at the correct place in alphabetic ascending order. If the list is out of order, it does its best to find the correct place, but results may be unpredictable.

Examples

This example populates the fourth level of a TreeView control:

```
long ll_lev1, ll_lev2, ll_lev3, ll_lev4
int  index

tv_list.PictureHeight = 32
tv_list.PictureWidth = 32

ll_lev1 = tv_list.InsertItemLast(0,"Composers",1)
ll_lev2 = tv_list.InsertItemLast(ll_lev1,&
   "Beethoven",2)
ll_lev3 = tv_list.InsertItemLast(ll_lev2,&
   "Symphonies",3)
FOR index = 1 to 9
   ll_lev4 = tv_list.InsertItemSort(ll_lev3, &
      "Symphony # " + String(index), 4)
NEXT

tv_list.ExpandItem(ll_lev3)
tv_list.ExpandItem(ll_lev4)
```

See also

InsertItem
InsertItemLast
InsertItemFirst

Syntax 2

For TreeView controls

Description

Inserts an item in sorted order, if possible.

Applies to

TreeView controls

Syntax

treeviewname.**InsertItemSort** (*handleparent, item*)

Argument	Description
treeviewname	The TreeView control in which you want to sort an item as a child of its parent, according to its label
handleparent	The handle of the item that will be the inserted item's parent. To insert the item at the first level, specify 0
item	A TreeViewItem structure for the item you are inserting

Return value

Long. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.

Usage

If SortType is anything except Unsorted!, the TreeView is always in a sorted state and you don't need to use InsertItemSort—you can use any insert function.

If SortType is Unsorted!, InsertItemSort attempts to insert the item at the correct place in alphabetic ascending order. If the list is out of order, it does its best to find the correct place, but results may be unpredictable.

Examples

This example inserts the current item beneath the root item in a TreeView control and sorts it according to its label:

```
long ll_handle, ll_roothandle
treeviewitem l_tvi

ll_handle = tv_list.FindItem(CurrentTreeItem!, 0)
ll_roothandle = tv_list.FindItem(RootTreeItem!, 0)
tv_list.GetItem(ll_handle , l_tvi)

tv_list.InsertItemSort(ll_roothandle, l_tvi)
```

See also

InsertItem
InsertItemFirst
InsertItemLast

InsertObject

Description

Displays the standard Insert Object dialog box, allowing the user to choose a new or existing OLE object, and inserts the selected object in the OLE control.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Syntax

olecontrol.**InsertObject** ( )

Argument	Description
olecontrol	The name of the OLE 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 Error

If any argument's value is NULL, InsertObject returns NULL.

Examples

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

InsertPicture

Description

Inserts a bitmap at the insertion point in a RichTextEdit control.

Platform information
This function has no effect on Macintosh.

Applies to

RichTextEdit controls

Syntax

rtename.**InsertPicture** (*filename*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to insert a picture
filename	A string whose value is the name of the file that contains the bitmap

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *filename* is NULL, InsertPicture returns NULL.

Usage

If there is a selection, InsertPicture inserts the bitmap at the beginning of the selection. The bitmap and the selection remain selected.

Examples

This example inserts a BMP file at the insertion point in the RichTextEdit control rte_1:

```
integer li_rtn
li_rtn = rte_1.InsertPicture("c:\windows\earth.bmp")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
li_rtn = rte_1.InsertPicture("clipart/earth.bmp")
```

See also

InputFieldInsert
InsertDocument

InsertRow

Description

Inserts a row in a DataWindow or DataStore. If any columns have default values, the row is initialized with these values before it is displayed.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**InsertRow** (*row*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If any argument's value is NULL, InsertRow returns NULL.

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.

A newly inserted row (with a status flag of New!) is not included in the modified count until data is entered in the row (its status flag becomes NewModified!).

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*)

Argument	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. If any argument's value is NULL, InsertSeries returns NULL.

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*)

Argument	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*. If *n* is too small or too large to be represented as an integer, Int returns 0. If *n* is NULL, Int returns NULL.

Usage

When the result for Int would be smaller than -32768 or larger than 32767, Int returns 0 because the result cannot be represented as an integer.

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 the *DataWindow Reference*

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*)

Argument	Description
stringorblob	A 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. *Stringorblob* can also be an Any variable containing a string or blob

Return value

Integer. Returns the value of *stringorblob* as an integer if it succeeds and 0 if *stringorblob* is not a valid number or is an incompatible data type. If *stringorblob* is NULL, Integer returns NULL.

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, this 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 the *DataWindow Reference*

InternetData

Description

Processes the HTML data returned by a GetURL or PostURL function. The Context object calls this function; you do not call this function explicitly. Instead, you override this function in a customized descendant of the InternetRequest standard class user object.

Applies to

InternetRequest objects

Syntax

servicereference.**InternetData** (*data*)

Argument	Description
servicereference	Reference to the Internet service instance
data	Blob containing the complete data requested by a GetURL or PostURL function

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Override this function in a user object that is a descendant of InternetRequest. The overridden function must contain one argument of type blob, which is passed by value. It should return an integer, processing *data* as appropriate for the situation.

Do not call this function explicitly
Do not code calls to this function. The GetURL and PostURL functions include an argument that references an instantiated InternetRequest descendant. When these functions complete, the Context object calls the InternetData function, returning HTML in *data*.

Examples

This example shows code you might use in an overridden InternetData function to display data from a GetURL function:

```
MessageBox("HTML from GetURL", String(data))
RETURN 1
```

See also

GetURL
PostURL

IntHigh

Description

Returns the high word of a long value.

Syntax

IntHigh (*long*)

Argument	Description
long	A long value

Return value

Integer. Returns the high word of *long* if it succeeds and -1 if an error occurs. If *long* is NULL, IntHigh returns NULL.

Usage

One use for IntHigh is for decoding values returned by external C functions and Windows messages.

Examples

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*)

Argument	Description
long	A long value

Return value

Integer. Returns the low word of *long* if it succeeds and -1 if an error occurs. If *long* is NULL, IntLow returns NULL.

Usage

One use for IntLow is for decoding values returned by external C functions and Windows messages.

Examples

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

InvokePBFunction

Description

Invokes the specified user-defined window function in the child window contained in a PowerBuilder window ActiveX control.

Applies to

Window ActiveX controls

Syntax

activexcontrol.**InvokePBFunction** (*name* {, *numarguments* {, *arguments* } })

Argument	Description
activexcontrol	Identifier for the instance of the PowerBuilder Window ActiveX control. When used in HTML, this is the NAME attribute of the object element. When used in other environments, this references the control that contains the PowerBuilder window ActiveX
name	String specifying the name of the user-defined window function. This argument is passed by reference
numarguments (optional)	Integer specifying the number of elements in the *arguments* array. The default is zero
arguments (optional)	Variant array containing function arguments. In PowerBuilder, Variant maps to the Any data type. This argument is passed by reference If you specify this argument, you must also specify *numarguments*. If you do not specify this argument and the function contains arguments, populate the argument list by calling the SetArgElement function once for each argument JavaScript cannot use this argument

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to invoke a user-defined window function in the child window contained in a PowerBuilder window ActiveX control.

To check the PowerBuilder function's return value, call the GetLastReturn function.

JavaScript cannot use the *arguments* argument.

Examples

This JavaScript example calls the InvokePBFunction function:

```
function invokeFunc(f) {
   var retcd;
   var rc;
   var numargs;
   var theFunc;
```

```
    var theArg;
    retcd = 0;
    numargs = 1;
    theArg = f.textToPB.value;
    PBRX1.SetArgElement(1, theArg);
    theFunc = "of_args";
    retcd = PBRX1.InvokePBFunction(theFunc, numargs);
    rc = parseInt(PBRX1.GetLastReturn());
    IF (rc != 1) {
       alert("Error. Empty string.");
    }
    PBRX1.ResetArgElements();
}
```

This VBScript example calls the InvokePBFunction function:

```
Sub invokeFunction_OnClick()
    Dim retcd
    Dim myForm
    Dim args(1)
    Dim rc
    Dim numargs
    Dim theFunc
    Dim rcfromfunc
    retcd = 0
    numargs = 1
    rc = 0
    theFunc = "of_args"
    Set myForm = Document.buttonForm
    args(0) = buttonForm.textToPB.value
    retcd = PBRX1.InvokePBFunction(theFunc, &
       numargs, args)
    rc = PBRX1.GetLastReturn()
    IF rc <> 1 THEN
       msgbox "Error. Empty string."
    END IF
    PBRX1.ResetArgElements()
END sub
```

See also

GetLastReturn
SetArgElement
TriggerPBEvent

IsAllArabic

Description

Tests whether a particular string is composed entirely of Arabic characters.

Platform information
The Arabic functions are available only in Arabic-enabled PowerBuilder.

Syntax

IsAllArabic (*string*)

Argument	Description
string	A string whose value you want to test to find out if it is composed entirely of Arabic characters

Return value

Boolean. Returns TRUE if *string* is composed entirely of Arabic characters and FALSE if it is not. The presence of numbers, spaces, and punctuation marks will also result in a return value of FALSE.

Usage

If you are not running Arabic-enabled PowerBuilder, IsAllArabic is set to FALSE.

Examples

Under Arabic-enabled PowerBuilder, this statement returns TRUE if the SingleLineEdit sle_name is composed entirely of Arabic characters:

```
IsAllArabic(sle_name.Text)
```

See also

IsAnyArabic
IsArabic
IsArabicAndNumbers
Reverse

IsAllHebrew

Description

Tests whether a particular string is composed entirely of Hebrew characters.

Platform information
The Hebrew functions are available only in Hebrew-enabled PowerBuilder.

Syntax

IsAllHebrew (*string*)

Argument	Description
string	A string whose value you want to test to find out if it is composed entirely of Hebrew characters

Return value

Boolean. Returns TRUE if *string* is composed entirely of Hebrew characters and FALSE if it is not. The presence of numbers, spaces, and punctuation marks will also result in a return value of FALSE.

Usage

If you are not running Hebrew-enabled PowerBuilder, IsAllHebrew is set to FALSE.

Examples

Under Hebrew-enabled PowerBuilder, this statement returns TRUE if the SingleLineEdit sle_name is composed entirely of Hebrew characters:

```
IsAllHebrew(sle_name.Text)
```

See also

IsAnyHebrew
IsHebrew
IsHebrewAndNumbers
Reverse

IsAnyArabic

Description

Tests whether a particular string contains at least one Arabic character.

Platform information
The Arabic functions are available only in Arabic-enabled PowerBuilder.

Syntax

IsAnyArabic (*string*)

Argument	Description
string	A string whose value you want to test to find out if it contains at least one Arabic character

Return value

Boolean. Returns TRUE if *string* contains at least one Arabic character and FALSE if it does not.

Usage

If you are not running Arabic-enabled PowerBuilder, IsAnyArabic is set to FALSE.

Examples

Under Arabic-enabled PowerBuilder, this statement returns TRUE if the SingleLineEdit sle_name contains at least one Arabic character:

```
IsAnyArabic(sle_name.Text)
```

See also

IsAllArabic
IsArabic
IsArabicAndNumbers
Reverse

IsAnyHebrew

Description

Tests whether a particular string contains at least one Hebrew character.

Platform information
The Hebrew functions are available only in Hebrew-enabled PowerBuilder.

Syntax

IsAnyHebrew (*string*)

Argument	Description
string	A string whose value you want to test to find out if it contains at least one Hebrew character

Return value

Boolean. Returns TRUE if *string* contains at least one Hebrew character and FALSE if it does not.

Usage

If you are not running Hebrew-enabled PowerBuilder, IsAnyHebrew is set to FALSE.

Examples

Under Hebrew-enabled PowerBuilder, this statement returns TRUE if the SingleLineEdit sle_name contains at least one Hebrew character:

```
IsAnyHebrew(sle_name.Text)
```

See also

IsAllHebrew
IsHebrew
IsHebrewAndNumbers
Reverse

IsArabic

Description

Tests whether a particular character is an Arabic character. For a string, IsArabic tests only the first character on the left.

Platform information
The Arabic functions are available only in Arabic-enabled PowerBuilder.

Syntax

IsArabic (*character*)

Argument	Description
character	A character or string whose value you want to test to find out if it is an Arabic character.

Return value

Boolean. Returns TRUE if *character* is an Arabic character and FALSE if it is not.

Usage

If you are not running Arabic-enabled PowerBuilder, IsArabic is set to FALSE.

Examples

Under Arabic-enabled PowerBuilder, this statement returns TRUE if the SingleLineEdit sle_name begins with an Arabic character:

```
IsArabic(sle_name.Text)
```

See also

IsAllArabic
IsAnyArabic
IsArabicAndNumbers
Reverse

IsArabicAndNumbers

Description

Tests whether a particular string is composed entirely of Arabic characters or numbers.

Platform information
The Arabic functions are available only in Arabic-enabled PowerBuilder.

Syntax

IsArabicAndNumbers (*string*)

Argument	Description
string	A string whose value you want to test to find out if it is composed entirely of Arabic characters or numbers

Return value

Boolean. Returns TRUE if *string* is composed entirely of Arabic characters or numbers and FALSE if it is not.

Usage

If you are not running Arabic-enabled PowerBuilder, IsArabicAndNumbers is set to FALSE.

Examples

Under Arabic-enabled PowerBuilder, this statement returns TRUE if the SingleLineEdit sle_name is composed entirely of Arabic characters and numbers:

```
IsArabicAndNumbers(sle_name.Text)
```

See also

IsAllArabic
IsAnyArabic
IsArabic
Reverse

IsDate

Description

Tests whether a string value is a valid date.

Syntax

IsDate (*datevalue*)

Argument	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. If *datevalue* is NULL, IsDate returns NULL.

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 the *DataWindow Reference*

IsHebrew

Description

Tests whether a particular character is a Hebrew character. For a string, IsHebrew tests only the first character on the left.

Platform information
The Hebrew functions are available only in Hebrew-enabled PowerBuilder.

Syntax

IsHebrew (*character*)

Argument	Description
character	A character or string whose value you want to test to find out if it is an Hebrew character

Return value

Boolean. Returns TRUE if *character* is an Hebrew character and FALSE if it is not.

Usage

If you are not running Hebrew-enabled PowerBuilder, IsHebrew is set to FALSE.

Examples

Under Hebrew-enabled PowerBuilder, this statement returns TRUE if the SingleLineEdit sle_name begins with a Hebrew character:

```
IsHebrew(sle_name.Text)
```

See also

IsAllHebrew
IsAnyHebrew
IsHebrewAndNumbers
Reverse

IsHebrewAndNumbers

Description

Tests whether a particular string is composed entirely of Hebrew characters and numbers.

Platform information
The Hebrew functions are available only in Hebrew-enabled PowerBuilder.

Syntax

IsHebrewAndNumbers (*string*)

Argument	Description
string	A string whose value you want to test to find out if it is composed entirely of Hebrew characters and numbers

Return value

Boolean. Returns TRUE if *string* is composed entirely of Hebrew characters and numbers and FALSE if it is not.

Usage

If you are not running Hebrew-enabled PowerBuilder, IsHebrewAndNumbers is set to FALSE.

Examples

Under Hebrew-enabled PowerBuilder, this statement returns TRUE if the SingleLineEdit sle_name is composed entirely of Hebrew characters and numbers:

```
IsHebrewAndNumbers(sle_name.Text)
```

See also

IsAllHebrew
IsAnyHebrew
IsHebrew
Reverse

IsNull

Description

Reports whether the value of a variable or expression is NULL.

Syntax

IsNull (*any*)

Argument	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.

Setting a variable to NULL
To set a variable to NULL, use the SetNull function.

Examples

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 the *DataWindow Reference*

IsNumber

Description

Reports whether the value of a string is a number.

Syntax

IsNumber (*string*)

Argument	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. If *string* is NULL, IsNumber returns NULL.

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 the *DataWindow Reference*

IsPreview

Description

Reports whether a RichTextEdit control is in preview mode.

Applies to

RichTextEdit controls

Syntax

rtename.**IsPreview** ( )

Argument	Description
rtename	The name of the RichTextEdit control for which you want to know whether it is in preview mode

Return value

Boolean. Returns TRUE if *rtename* is in preview mode and FALSE if it is in data entry mode.

Examples

This example switches the RichTextEdit control rte_1 to preview mode if it isn't already and then prints it:

```
IF NOT rte_1.IsPreview() THEN
    rte_1.Preview(TRUE)
    rte_1.Print(1, "1-4", FALSE, TRUE)
END IF
```

See also

Preview

IsSelected

Description

Determines whether a row is selected in a DataWindow or DataStore. A selected row is highlighted using reverse video.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**IsSelected** (*row*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol* is selected and FALSE if it is not selected. If *row* is greater than the number of rows in *dwcontrol* or is 0 or negative, IsSelected also returns FALSE. If any argument's value is NULL, IsSelected returns NULL.

Usage

You can call IsSelected in a script for the Clicked event to determine whether the row the user clicked was selected.

Examples

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
```

This 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 the *DataWindow Reference*

IsTime

Description

Reports whether the value of a string is a valid time value.

Syntax

IsTime (*timevalue*)

Argument	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. If *timevalue* is NULL, IsTime returns NULL.

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 the *DataWindow Reference*

IsValid

Description

Determines whether an object variable is instantiated—whether its value is a valid object handle.

Syntax

IsValid (*objectname*)

Argument	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.

Examples

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*)

Argument	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. (Not all ASCII values are recognized; see Usage.) A table of KeyCode values follows Examples

Return value

Boolean. Returns TRUE if *keycode* was pressed and FALSE if it was not. If *keycode* is NULL, KeyDown returns NULL.

Usage

KeyDown does not report what character the user typed—it reports whether the user was pressing the specified key when the event whose script is calling KeyDown was triggered.

Events You can call KeyDown in a window's Key event or a keypress event for a control 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. The Key event is triggered repeatedly if the user holds down a repeating key. For controls, you can define a user event for pbm_keydown or pbm_dwnkey (DataWindows), and call KeyDown in its script.

You can also call KeyDown in a mouse event, such as Clicked, to determine whether the user also pressed a modifier key, such as CTRL.

KeyCodes and ASCII values KeyDown does not distinguish between uppercase and lowercase letters or other characters and their shifted counterparts. For example, KeyA! refers to the A key—the user may have typed "A" or "a." Key9! refers to both "9" and "(". Instead, you can test whether a modifier key is also pressed.

KeyDown does not test whether CAPS LOCK or other toggle keys are in a toggled-on state, only whether the user is pressing it.

KeyDown only detects ASCII values 65-90 (KeyA! - KeyZ!) and 48-57 (Key0!-Key9!). These ASCII values detect whether the key was pressed, whether or not the user also pressed SHIFT or CAPS LOCK. KeyDown does not detect other ASCII values (such as 97-122 for lowercase letters).

The following table categorizes KeyCode values by type of key and provides explanations of names that might not be obvious.

Type of key	KeyCode values and descriptions
Mouse buttons	KeyLeftButton! Left mouse button KeyMiddleButton! Middle mouse button KeyRightButton! Right mouse button
Letters	KeyA! - KeyZ! A - Z, uppercase or lowercase
Other symbols	KeyQuote! ' and " KeyEqual! = and + KeyComma! , and < KeyDash! - and _ KeyPeriod! . and > KeySlash! / and ? KeyBackQuote! ` and ~ KeyLeftBracket! [and { KeyBackSlash! \ and | KeyRightBracket!] and } KeySemiColon! ; and :
Non-printing characters	KeyBack! Backspace KeyTab! KeyEnter! KeySpaceBar!
Function keys	KeyF1! - KeyF12! Function keys F1 to F12
Control keys	KeyShift! KeyControl! KeyAlt! KeyPause! KeyCapsLock! KeyEscape! KeyPrintScreen! KeyInsert! KeyDelete!
Navigation keys	KeyPageUp! KeyPageDown! KeyEnd! KeyHome! KeyLeftArrow! KeyUpArrow! KeyRightArrow! KeyDownArrow!

Type of key	KeyCode values and descriptions
Numeric and symbol keys	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 (
Keypad numbers	KeyNumpad0! - KeyNumpad9! 0 - 9 on the numeric keypad
Keypad symbols	KeyMultiply! * on numeric keypad KeyAdd! + on numeric keypad KeySubtract! - on numeric keypad KeyDecimal! . on numeric keypad KeyDivide! / on numeric keypad KeyNumLock! KeyScrollLock!

Examples

The following code checks whether the user pressed the 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 TAB, ENTER, or any of the scrolling keys:

```
IF (KeyDown(KeyTab!) OR KeyDown(KeyEnter!) OR &
KeyDown(KeyDownArrow!) OR KeyDown(KeyUpArrow!) &
OR KeyDown(KeyPageDown!) OR KeyDown(KeyPageUp!))&
THEN
```

This statement tests whether the user pressed the A key (ASCII value 65):

```
IF KeyDown(65) THEN ...
```

This statement tests whether the user pressed the SHIFT key and the A key:

```
IF KeyDown(65) AND KeyDown(KeyShift!) THEN ...
```

This statement in a Clicked event checks whether the SHIFT is also pressed:

```
IF KeyDown(KeyShift!) THEN ...
```

Left

Description

Obtains a specified number of characters from the beginning of a string.

Syntax

Left (*string*, *n*)

Argument	Description
string	The string containing the characters you want
n	A long specifying the number of characters you want

Usage

String. Returns the leftmost *n* characters in *string* if it succeeds and the empty string ("") if an error occurs. If any argument's value is NULL, Left returns NULL.

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 the *DataWindow Reference*

LeftTrim

Description

Removes spaces from the beginning of a string.

Syntax

LeftTrim (*string*)

Argument	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. If *string* is NULL, LeftTrim returns NULL.

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 the *DataWindow Reference*

Len

Description

Reports the length of a string or a blob.

Syntax

Len (*stringorblob*)

Argument	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. If *stringorblob* is NULL, Len returns NULL.

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 the *DataWindow Reference*

Length

Description

Reports the length in bytes of an open OLE stream.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

oleStream objects

Syntax

olestream.**Length** (*sizevar*)

Argument	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

If any argument's value is NULL, Length returns NULL.

Examples

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 function's 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* })

Argument	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. If any argument's value is NULL, LibraryCreate returns NULL.

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.

Examples

This statement in Windows 3.1 or Windows NT creates a library named dwTemp in the pb directory on drive C and associates a comment with the library:

```
LibraryCreate("c:\pb\dwTemp.pbl", &
"Temporary library for dynamic DataWindows")
```

This statement creates the same library in the PowerBuilder folder on a Macintosh drive:

```
LibraryCreate("HardDisk:PowerBuilder:dwTemp.pbl",&
"Temporary library for dynamic DataWindows")
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
LibraryCreate("HardDisk:PowerBuilder:dwTemp.pbl ", &
"Temporary library for dynamic DataWindows")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
LibraryCreate("/export/home/pb/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* })

Argument	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. If any argument's value is NULL, LibraryDelete returns NULL.

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:\pb\dwTemp.pbl", "d_emp", &
ImportDataWindow!)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
LibraryDelete("HD:PB Libraries:dwTemp.pbl", &
"d_emp", ImportDataWindow!)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
LibraryDelete("/export/home/pb/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*)

Argument	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
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. If any argument's value is NULL, LibraryDirectory returns NULL.

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 INFO For an example of parsing tab-delimited data, see the Pos function.

Examples

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:\pb\dwTemp.pbl", DirUserObject!)
dw_list.SetRedraw(FALSE)
dw_list.Reset( )
dw_list.ImportString(ls_Entries)
dw_list.SetRedraw(TRUE)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
ls_entries = LibraryDirectory( &
"HD:PB Libraries:dwTemp.pbl", DirUserObject!)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
ls_entries = LibraryDirectory( &
"/export/home/pb/dwtemp.pbl", DirUserObject!)
```

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*)

Argument	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
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. If any argument's value is NULL, LibraryExport returns NULL.

Examples

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:\pb\dwTemp.pbl", &
"d_emp", ExportDataWindow!)
dw_1.Create(ls_dwsyn, ls_errors)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
ls_dwsyn = LibraryExport( &
"HD:PBLibraries:dwTemp.pbl", &
"d_emp", ExportDataWindow!)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
ls_dwsyn = LibraryExport( &
"/export/home/pb/dwtemp.pbl", &
"d_emp", ExportDataWindow!)
```

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* })

Argument	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 import. The only supported object type is ImportDataWindow!
syntax	A string specifying the syntax of the DataWindow object you want to import
errors	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. If any argument's value is NULL, LibraryImport returns NULL.

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:\pb\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:\pb\dwTemp.pbl", &
"d_emp", ImportDataWindow!, &
dwsyntax, ErrorBuffer, &
"Employee DataWindow 1")
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
rtncode = LibraryImport("HD:PB Libraries:dwTemp.pbl",
&
"d_emp", ImportDataWindow!, &
dwsyntax, ErrorBuffer, &
"Employee DataWindow 1")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
rtncode = LibraryImport
("/export/home/pb/dwtemp.pbl", &
"d_emp", ImportDataWindow!, &
dwsyntax, ErrorBuffer, &
"Employee DataWindow 1")
```

See also

Describe
LibraryCreate
LibraryDelete
LibraryDirectory
LibraryImport

LineCount

Description

Determines the number of lines in an edit control that allows multiple lines.

Applies to

RichTextEdit, MultiLineEdit, EditMask, and DataWindow controls

Syntax

editname.**LineCount** ( )

Argument	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, or RichTextEdit for which you want the number of lines

Return value

Long. Returns the number of lines in *editname* if it succeeds and -1 if an error occurs. If *editname* is NULL, LineCount returns NULL.

Usage

LineCount counts each visible line, whether it was the result of wrapping or carriage returns.

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 property is TRUE. If horizontal scrolling is enabled with these properties, the user must press enter to type additional lines.

A RichTextEdit control always contains an end-of-file mark even if there is no text in the control. Therefore, its line count is always at least 1. Other edit controls, when empty, have a line count of 0.

Examples

If the MultiLineEdit mle_Instructions has 9 lines, this 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 an edit control.

Applies to

RichTextEdit, MultiLineEdit, and EditMask controls

Syntax

editname.**LineLength** ( )

Argument	Description
editname	The name of the RichTextEdit, MultiLineEdit, or EditMask in which you want to determine the length of the line containing the insertion point

Return value

Long. Returns the length of the line containing the insertion point in *editname*. Returns -1 if an error occurs. If *editname* is NULL, LineLength returns NULL.

Usage

If the control contains a selection instead of a single insertion point, LineLength counts the line at the beginning of the selection.

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.

Examples

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

LineList

Description

Provides a list of the lines in a routine included in a performance analysis model.

Applies to

ProfileRoutine object

Syntax

iinstancename.**LineList** (*list*)

Argument	Description
instancename	Instance name of the ProfileRoutine object
list	An unbounded array variable of data type ProfileLine in which LineList stores a ProfileLine object for each line in the routine. This argument is passed by reference

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- ModelNotExistsError!—The model does not exist

Usage

Use this function to extract a list of the lines in a routine included in the performance analysis model. You must have previously created the performance analysis model from a trace file using the BuildModel function. Each line is defined as a ProfileLine object and provides the number of times the line was hit, any calls made from the line, and the time spent on the line and in any called functions. The lines are listed in numeric order.

Lines are not returned for database statements and objects. If line information was not logged in the trace file, lines are not returned.

Examples

This example gets a list of the routines included in a performance analysis model and then gets a list of the lines in each routine:

```
Long ll_cnt
ProfileLine lproln_line[]

lpro_model.BuildModel()
lpro_model.RoutineList(iprort_list)

FOR ll_cnt = 1 TO UpperBound(iprort_list)
   iprort_list[ll_cnt].LineList(lproln_line)
   ...
NEXT
```

See also

BuildModel

LinkTo

Description

Establishes a link between an OLE control and a file or an item within the file.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Syntax

olecontrol.**LinkTo** (*filename* {, *sourceitem* })

Argument	Description
olecontrol	The name of the OLE 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 *olecontrol*, 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

If any argument's value is NULL, LinkTo returns NULL.

Examples

This example creates an object in the OLE 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

Listen

Description

Instructs a server application to begin listening for client connections. Client applications cannot connect to a server application until the server has executed the Listen function.

This function applies to distributed applications only.

Applies to

Transport objects

Syntax

transport.**Listen** ( )

Argument	Description
transport	The name of the Transport object you want to use to process client requests for connections. The Transport object has properties that specify how the server application will handle client requests

Return value

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

50 Distributed service error
52 Distributed communications error

Usage

Before calling Listen, you need to assign values to the properties of the Transport object. The properties you set vary depending on which communications driver you are using.

The Listen function is not supported with the Local driver.

Examples

In this example, the server application uses the mytransport Transport object to begin listening for client connections:

```
transport mytransport
mytransport = create transport
mytransport.driver = "WinSock"
mytransport.application = "dpbserv"
mytransport.Listen()
```

See also

StopListening

Log

Description

Determines the natural logarithm of a number.

Syntax

Log (*n*)

Argument	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. If *n* is NULL, Log returns NULL.

Inverse of Log
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 the *DataWindow Reference*

LogTen

Description

Determines the base 10 logarithm of a number.

Syntax

LogTen (*n*)

Argument	Description
n	The number for which you want the base 10 logarithm. The value of *n* must not be negative

Usage

Double. Returns the base 10 logarithm of *n*. An execution error occurs if *n* is negative. If *n* is NULL, LogTen returns NULL.

Inverse of LogTen 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
LogTen
LogTen in the *DataWindow Reference*

Long

Converts data into data of type long. There are two syntaxes.

To	Use
Combine two unsigned integers into a long value	Syntax 1
Convert a string whose value is a number into a long or to obtain a long value stored in a blob	Syntax 2

Syntax 1

For combining integers

Description

Combines two unsigned integers into a long value.

Syntax

Long (*lowword*, *highword*)

Argument	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

Long. Returns the long if it succeeds and -1 if an error occurs. If any argument's value is NULL, Long returns NULL.

Usage

Use Long for passing values to external C functions or specifying a value for the LongParm property of PowerBuilder's Message object.

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)
```

Syntax 2 For converting strings and blobs

Description

Converts a string whose value is a number into a long or obtains a long value stored in a blob.

Syntax

Long (*stringorblob*)

Argument	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. *Stringorblob* can also be an Any variable containing a string or blob

Return value

Long. Returns the value of *stringorblob* as a long if it succeeds and 0 if *stringorblob* is not a valid PowerScript number or if it is an incompatible data type. If *stringorblob* is NULL, Long returns NULL.

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 Long function.

Examples

This statement returns 2167899876 as a long:

```
Long("2167899876")
```

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 the *DataWindow Reference*

Lower

Description

Converts all the characters in a string to lowercase.

Syntax

Lower (*string*)

Argument	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. If *string* is NULL, Lower returns NULL.

Examples

This statement returns babe ruth:

```
Lower("Babe Ruth")
```

See also

Upper
Lower in the *DataWindow Reference*

LowerBound

Description

Obtains the lower bound of a dimension of an array.

Syntax

LowerBound (*array* {, *n* })

Argument	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

Long. Returns the lower bound of dimension *n* of *array* and -1 if *n* is greater than the number of dimensions of the array. If any argument's value is NULL, LowerBound returns NULL.

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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailAddress** ({ *mailmessage* })

Argument	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 an Address dialog box

Return value

mailReturnCode. Returns one of the following values:

mailReturnSuccess!
mailReturnFailure!
mailReturnInsufficientMemory!
mailReturnUserAbort!

If any argument's value is NULL, mailAddress returns NULL.

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 (such as 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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailDeleteMessage** (*messageid*)

Argument	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!

If any argument's value is NULL, mailDeleteMessage returns NULL.

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.

Examples

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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailGetMessages** ({ *messagetype*, } { *unreadonly* })

Argument	Description
mailsession	A mailSession 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!

If any argument's value is NULL, mailGetMessages returns NULL.

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. With mailReadMessage, for example, 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.

Examples

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
long n, 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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailHandle** ( )

Argument	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. If *mailsession* is NULL, mailHandle returns NULL.

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.

Examples

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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailLogoff** ( )

Argument	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.

Examples

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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailLogon** ({ *userid*, *password* } {, *logonoption* })

Argument	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
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!

If any argument's value is NULL, mailLogon returns NULL.

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, and so on) 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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailReadMessage** (*messageid*, *mailmessage*, *readoption*, *mark*)

Argument	Description
mailsession	A mailSession object identifying the session in which you want to read a message
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 ♦ mailSuppressAttachments! — 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!

If any argument's value is NULL, mailReadMessage returns NULL.

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 property of the mailMessage object. The AttachmentFile property is a mailFileDescription object. Its PathName property 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.

Examples

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. Assuming 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 supplied with PowerBuilder.

See also

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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailRecipientDetails** (*mailrecipient* {, *allowupdates* })

Argument	Description
mailsession	A mailSession 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!
mailReturnUnknownRecipient!
mailReturnUserAbort!

If any argument's value is NULL, mailRecipientDetails returns NULL.

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.

Examples

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 a property 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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailResolveRecipient** (*recipient* {, *allowupdates* })

Argument	Description
mailsession	A mailSession object identifying the session in which you want to resolve the recipient
recipient	A mailRecipient structure or a string variable whose value is a recipient's name. The recipient's name is a property 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!

If any argument's value is NULL, mailResolveRecipient returns NULL.

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, such as 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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailSaveMessage** (*messageid*, *mailmessage*)

Argument	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!

If any argument's value is NULL, mailSaveMessage returns NULL.

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 Macintosh or UNIX.

Applies to

mailSession object

Syntax

mailsession.**mailSend** ({ *mailmessage* })

Argument	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!

If any argument's value is NULL, mailSend returns NULL.

Usage

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 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 mailMessage 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 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*)

Argument	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. If any argument's value is NULL, Match returns NULL.

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.

A text pattern consists of **metacharacters**, which have special meaning in the match string, and **nonmetacharacters**, which match the characters themselves.

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 $

Metacharacter	Meaning	Example
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

Sample patterns 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)

This pattern	Matches
^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

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 the *DataWindow Reference*

Max

Description

Determines the larger of two numbers.

Syntax

Max (*x*, *y*)

Argument	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. If any argument's value is NULL, Max returns NULL.

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 the *DataWindow Reference*

MemberDelete

Description

Deletes a member from an OLE object in a storage. The member can be another OLE object (a substorage) or a stream.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLEStorage objects

Syntax

olestorage.**MemberDelete** (*membername*)

Argument	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

If any argument's value is NULL, MemberDelete returns NULL.

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.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLEStorage objects

Syntax

olestorage.**MemberExists** (*membername*, *exists*)

Argument	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

If any argument's value is NULL, MemberExists returns NULL.

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.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLEStorage objects

Syntax

olestorage.**MemberRename** (*membername*, *newname*)

Argument	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

If any argument's value is NULL, MemberRename returns NULL.

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* } } })

Argument	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. If any argument's value is NULL, MessageBox returns NULL.

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 you are running RightToLeft versions of PowerBuilder and Windows and want to display Arabic or Hebrew text for the message and buttons, set the RightToLeft property of the application object to TRUE. The characters of the message will display from right to left. However, the button text will continue to display in English unless you are running a localized version of PowerBuilder.

When MessageBox doesn't work

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, unexpected results can occur.

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.

For details, see PrintOpen.

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* })

Argument	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. If any argument's value is NULL, Mid returns NULL.

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, 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 the *DataWindow Reference*

Min

Description

Determines the smaller of two numbers.

Syntax

Min (*x*, *y*)

Argument	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. If any argument's value is NULL, Min returns NULL.

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 the *DataWindow Reference*

Minute

Description

Obtains the number of minutes in the minutes portion of a time value.

Syntax

Minute (*time*)

Argument	Description
time	The time value from which you want the minutes

Return value

Integer. Returns the minutes portion of *time* (00 to 59). If *time* is NULL, Minute returns NULL.

Examples

This statement returns 1:

```
Minute(19:01:31)
```

See also

Hour
Second
Minute in the *DataWindow Reference*

Mod

Description

Obtains the remainder (modulus) of a division operation.

Syntax

Mod (*x*, *y*)

Argument	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. If any argument's value is NULL, Mod returns NULL.

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 the *DataWindow Reference*

ModifiedCount

Description

Reports the number of rows that have been modified but not updated in a DataWindow or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**ModifiedCount** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If *dwcontrol* is NULL, ModifiedCount returns NULL.

Usage

ModifiedCount reports the number of rows that are scheduled to be added or updated in the database table associated with a DataWindow or DataStore. This includes rows in the primary and filter buffers.

A newly inserted row (with a status flag of New!) is not included in the modified count until data is entered in the row (its status flag becomes NewModified!).

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 properties. You can add and remove objects from the DataWindow object by providing specifications for the objects.

FOR INFO For lists and explanations of DataWindow object properties, see the DataWindow Reference.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**Modify** (*modstring*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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*. If any argument's value is NULL, Modify returns NULL.

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 object'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 (such as 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 properties and values using the format you see in exported DataWindow syntax. To create an object, you must supply enough information to define it. *Object* cannot be an OLE object. You cannot add an OLE object to a DataWindow using the Modify function
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.property=value	Changes the value of *property* to *value*. Properties control the location, color, size, font, and other settings for *objectname*. When *objectname* is DataWindow, you can also set properties for database access. The *DataWindow Reference* has lists of objects in a DataWindow, their properties, and values. Depending on the specific property, *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 property. 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 properties of an object, give the object a name.

Expressions for Modify When you specify an expression for a DataWindow property, the expression has the format:

defaultvalue~t*DataWindowpainterexpression*

Defaultvalue is a value that can be converted to the appropriate data type for the property. 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 property. When you are setting a column's property, 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 properties accept expressions. For details on each property, see the *DataWindow Reference*.

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 INFO For more information about quotes and tildes, see "Standard data types" on page 24.

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 properties with a single call to Modify by including each property 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 property.

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 For efficiency, use Modify instead of SetSQLSelect to modify a WHERE clause. Modify is faster because it does not verify the syntax and does not change the update status of the DataWindow object. However, Modify is more susceptible to user error. 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 below in See also.

Modify is for modifying the properties of a DataWindow *object*. You can set properties 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 property 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=100,255,0))'")
```

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
```

This example sets the text color of a RadioButton column to the value of color1 (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 this 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 property:

```
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 SQL Anywhere and ORACLE, 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"
```

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 selection called Select Data. When the user chooses it, its script displays the DataWindow control in query mode and checks the menu item. 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 menu item 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 menu to show
        // 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 menu to show
            // 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 Menu 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 INFO For details on how you or the user specifies information in query mode, see the *PowerBuilder User's Guide*.

DataWindow presentation styles

You cannot use QueryMode and QuerySort with DataWindow objects that use any of the following presentation styles: N-Up, Label, Crosstab, RichText, and Graph.

Prompt for criteria is another way of letting the user specify retrieval criteria. You set it 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 properties that affect both query mode and prompt for criteria. For details, see the Criteria DataWindow object property in the *DataWindow Reference*.

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 property, 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 property 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
```

Replacing a DropDownDataWindow object
When you use Modify to replace one DropDownDataWindow object with another, for example:

```
dw_parent.Modify(dept_id.dddw.name= &
    d_dddw_empsal_by_dept )
```

PowerBuilder compares the column definitions in the two objects and reuses the original result set if the the number of columns and their data types match. The display and data value column names must exist in the data object SQL statements for both objects. If there are any differences, PowerBuilder will re-retrieve the data.

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:\PB\BEACH.BMP"
border="0" name=bmp1 )'

dw_cust.Modify(modstring)
```

Syntax for creating objects

To create an object you must provide DataWindow syntax. The easiest way to get correct syntax for all the necessary properties 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 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

Changes the value of a data point in a series on a graph. There are two syntaxes depending on the type of graph.

To modify a data point in	Use
All graph types except scatter	Syntax 1
Scatter graphs	Syntax 2

Syntax 1: for all graph types except scatter

Description

Changes the value of a data point in a series on a graph. You can specify the data point to be modified by position or by 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.**ModifyData** (*seriesnumber*, *datapoint*, *datavalue* {, *categoryvalue* })

Argument	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.

Usage

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 INFO 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)
```

See also

AddData
FindCategory
FindSeries
InsertCategory
InsertData

Syntax 2: for scatter graphs

Description

Changes the value of a data point in a series on a graph. 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

controlname.**ModifyData** (*seriesnumber*, *datapoint*, *xvalue*, *yvalue*)

Argument	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. If any argument's value is NULL, ModifyData returns NULL.

Usage

For scatter graphs, 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.

Examples

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
FindSeries

Month

Description

Determines the month of a date value.

Syntax

Month (*date*)

Argument	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*. If *date* is NULL, Month returns NULL.

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 the *DataWindow Reference*

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*)

Argument	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 or if *objectname* is a maximized window. If any argument's value is NULL, Move returns NULL.

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 (such as, 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.

Move does not move a maximized sheet or window. If the window is maximized, Move returns –1.

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 properties of the moved object.

Equivalent syntax The syntax below directly sets the X and Y properties 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 properties 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)
```

MoveTab

Description

Moves a tab page to another position in a Tab control, changing its index number.

Applies to

Tab controls

Syntax

tabcontrolname.**MoveTab** (*source, destination*)

Argument	Description
tabcontrolname	The name of the Tab control containing the tab you want to move.
source	An integer whose value is the index of the tab you want to move.
destination	An integer whose value is the index of the destination tab before which *source* is moved. If *destination* is 0 or greater than the number of tabs, *source* is moved to the end.

Return value

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

Usage

MoveTab also reorders the tab pages in the Tab control's Control array (which is a property that lists the tab pages within the Tab control) to match the new tab order.

Examples

This example moves the first tab to the end:

```
tab_1.MoveTab(1, 0)
```

This example move the fourth tab to the first position:

```
tab_1.MoveTab(4, 1)
```

This example move the fourth tab to the third position:

```
tab_1.MoveTab(4, 3)
```

See also

OpenTab
SelectTab

NextActivity

Description

Provides the next activity in a trace file.

Applies to

TraceFile objects

Syntax

instancename.**NextActivity** ( )

Argument	Description
instancename	Instance name of the TraceFile object

Return value

TraceActivityNode

Usage

You use the NextActivity function to read the next activity in a trace file. The activity is returned as a TraceActivityNode object. If there are no more activities or if the file is not open, an invalid object is returned. You can then use the LastError property of the TraceFile object to determine what kind of error occurred.

To use this function, you must have previously opened the trace file with the Open function. You use the NextActivity and Open functions as well as the other properties and functions provided by the TraceFile object to access the contents of a trace file directly. For example, you would use these functions if you want to perform your own analysis of the tracing data instead of using the available modeling objects.

Examples

This example opens a trace file and then uses a user-defined function called of_dumpactivitynode to report the appropriate information for each activity depending on its activity type:

```
String ls_filename, ls_line
TraceFile ltf_file
TraceActivityNode ltan_node

ls_filename = sle_filename.text
ltf_file = CREATE TraceFile
ltf_file.Open(ls_filename)

ls_line = "CollectionTime = " + &
   String(ltf_file.CollectionTime) + "~r~n" + &
   "Num Activities = " + &
   String(ltf_file.NumberOfActivities) + "~r~n
mle_output.text = ls_line

ltan_node = ltf_file.NextActivity()
```

```
DO WHILE IsValid(ltan_node)
    ls_line = of_dumpactivitynode(ltan_node)
    ltan_node = ltf_file.NextActivity()
    mle_output.text = ls_line
LOOP
```

See also

Open
Close
Reset

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 the *DataWindow Reference*

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*)

Argument	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) (optional)	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. If any argument's value is NULL, ObjectAtPointer also returns NULL.

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.

Call ObjectAtPointer first

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
grObjectTypeobject_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 Macintosh and UNIX.

Applies to

DataWindow controls and child DataWindows

Syntax

dwcontrol.**OLEActivate** (*row, column, verb*)

Argument	Description
dwcontrol	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. If any argument's value is NULL, OLEActivate returns NULL.

Usage

The user can activate OLE by double-clicking 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 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 SQL Anywhere, the 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.

FOR INFO For setup details, see *Application Techniques*.

Examples

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

Opens a window or an OLE object.

For windows Open displays a window and makes all its properties and controls available to scripts.

To	Use
Open an instance of a particular window data type	Syntax 1
Allow the application to select the window's data type when the script is executed	Syntax 2

For OLE objects Open loads an OLE object contained in a file or storage into an OLE control or storage object variable. The source and the target are then connected for the purposes of saving work.

To open	Use
An OLE object in a file and load it into an OLE control	Syntax 3
An OLE object in a storage object in memory and load it into an OLE control	Syntax 4
An OLE object in an OLE storage file and load it into a storage object in memory	Syntax 5
An OLE object that is a member of an open OLE storage and load it into a storage object in memory	Syntax 6
A stream in an OLE storage object in memory and load it into a stream object	Syntax 7

For trace files Open opens the specified trace file for reading.

To	Use
Open a trace file	Syntax 8

Syntax 1

For windows of a known data type

Description

Opens a window object of a known data type. Open displays the window and makes all its properties and controls available to scripts.

Applies to

Window objects

Syntax

Open (*windowvar* {, *parent* })

Argument	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) (optional)	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, Open returns NULL.

Usage

You must open a window before you can access the properties of the window. If you access the window's properties before you open it, an execution error will occur.

To reference an open window in scripts, use *windowvar*.

Calling Open twice
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.

To open an array of windows where each window has different data type, use Syntax 2 of Open.

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.

Mouse behavior and response windows
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 response window is opened while the mouse is captured, unexpected results can occur.

Because a response window grabs focus, you should not open it when focus is changing, such as in a LoseFocus event.

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)
```

See also

Close
OpenWithParm
Show

Syntax 2

For windows of unknown data type

Description

Opens a window object when you don't know its data type until the application is running. Open displays the window and makes all its properties and controls available to scripts.

Applies to

Window objects

Syntax

Open (*windowvar*, *windowtype* {, *parent* })

Argument	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) (optional)	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, Open returns NULL.

Usage

You must open a window before you can access the properties of the window. If you access the window's properties before you open it, an execution error will occur.

To reference an open window in scripts, use *windowvar*.

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 this syntax 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.

Considerations when specifying a window type

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 descendent window, you can only reference properties, 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 your application.

FOR INFO For information about the parent of an opened window, see Syntax 1.

Examples

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 array 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
```

See also

Close
OpenWithParm
Show

Syntax 3

For loading an OLE object from a file into a control

Description

Opens an OLE object in a file and loads it into an OLE control.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE controls

Syntax

olecontrol.**Open** (*OLEsourcefile*)

Argument	Description
olecontrol	The name of the OLE control into which you want to load an OLE object

Argument	Description
OLEsourcefile	A string specifying the name of an OLE storage file containing the object. The file must already exist and contain an OLE object. *OLEsourcefile* can include a path for the file, as well as path information inside the OLE storage

Return value

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

If any argument's value is NULL, Open returns NULL.

Examples

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")
```

See also

InsertFile
Save
SaveAs

Syntax 4

For opening an OLE object in memory into a control

Description

Opens an OLE object that is in a OLE storage object in memory and loads it into an OLE control.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE controls

Syntax

olecontrol.**Open** (*sourcestorage*, *substoragename*)

Argument	Description
olecontrol	The name of the OLE control into which you want to load an OLE object

Argument	Description
sourcestorage	The name of an object variable of OLEStorage containing the object you want to load into *olecontrol*
substoragename	A string specifying the name of a substorage that contains the desired object within *storagename*

Return value

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

If any argument's value is NULL, Open returns NULL.

Examples

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)
```

See also

InsertFile
Save
SaveAs

Syntax 5

For opening an OLE object in a file into an OLEStorage

Description

Opens an OLE object in an OLE storage file and loads it into a storage object in memory.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE storage objects

Syntax

olestorage.**Open** (*OLEsourcefile* {, *readmode* {, *sharemode* } })

Argument	Description
olestorage	The name of an 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 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 ♦ stgDenyRead! — Other attempts to open *OLEsourcefile* for reading will fail ♦ stgDenyWrite — Other attempts to open *OLEsourcefile* for writing will fail

Return value

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

-1 The file is not an OLE storage file
-3 The file is not found
-9 Other error

If any argument's value is NULL, Open returns NULL.

Usage

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, the PowerBuilder 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 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!)
```

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, using Syntax 6 of Open. (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
```

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 using Syntax 7 of Open. (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
Save
SaveAs

Syntax 6

For opening an OLE storage member into a storage

Description

Opens a member of an open OLE storage and loads it into another OLE storage object in memory.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE storage objects

Syntax

olestorage.**Open** (*substoragename*, *readmode*, *sharemode*, *sourcestorage*)

Argument	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 ♦ 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

Argument	Description
sourcestorage	An open OLEStorage object containing *substoragename*.

Return value

Return value

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

If any argument's value is NULL, Open returns NULL.

Usage

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 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
```

See also

Close
Save
SaveAs

Syntax 7

For opening OLE streams

Description

Opens a stream in an open OLE storage object and loads it into an OLE stream object.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE stream objects

Syntax

olestream.**Open** (*sourcestorage*, *streamname* {, *readmode* {, *sharemode* } })

Argument	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

Argument	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

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

If any argument's value is NULL, Open returns NULL.

Examples

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

Syntax 8

For opening trace files

Description

Opens the specified trace file for reading.

Applies to

TraceFile object

Syntax

instancename.**Open** (*filename*)

Argument	Description
instancename	Instancename of the TraceFile object
filename	A string identifying the name of the trace file you want to read

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileAlreadyOpenError!—The specified trace file has already been opened
- FileOpenError!—The trace file can not be opened for reading
- FileInvalidFormatError!—The file does not have the correct format
- EnterpriseOnlyFeature!—This function is supported only in the Enterprise edition of PowerBuilder
- SourcePBLError!—The source libraries cannot be found

Usage

You use this syntax to access the contents of a specified trace file created from a running PowerBuilder application. You can then use the properties and functions provided by the TraceFile object to perform your own analysis of tracing data instead of using the available modeling objects.

Examples

This example opens a trace file:

```
TraceFile ltf_file
String ls_filename

ltf_file = CREATE TraceFile
ltf_file.Open(ls_filename)
...
```

See also

Close
Reset
NextActivity

OpenChannel

Description

Opens a channel to a DDE server application.

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

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

OpenChannel (*applname*, *topicname* {, *windowhandle* })

Argument	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.

FOR INFO For more information, see StartServerDDE.

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* } })

Argument	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 value of 0 lists the open sheets under the next-to-last menu item
arrangeopen (optional)	A value of the ArrangeOpen 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. If any argument's value is NULL, OpenSheet returns NULL.

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.

PowerBuilder doesn't automatically copy objects that are dynamically referenced (through string variables) into your executable. To include the window object specified in *windowtype* in your application, list it in the resource (PBR) file that you use when you build the executable.

FOR INFO For more information about PBR files for an executable, see the *PowerBuilder User's Guide*.

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 menu selections. Make sure that the menu you specify or, if you leave out *position*, the next-to-last menu has at least one other item.

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.

Sheets in a frame cannot be made invisible. When you open a sheet, the value of the Visible property is ignored. Changing the Visible property when the window is already open has no effect.

Examples

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* } })

Argument	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
arrangeopen (optional)	A value of the ArrangeOpen 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. If any argument's value is NULL, OpenSheetWithParm returns NULL.

Usage

The system Message object has three properties 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 properties.

Message object property	Argument 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, which causes an error. 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.

FOR INFO See the usage notes for OpenSheet, which also apply to OpenSheetWithParm.

Examples

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

OpenTab

Opens a visual user object and makes it a tab page in the specified Tab control and makes all its properties and controls available to scripts.

To open	Use
A user object as a tab page	Syntax 1
A user object as a tab page, allowing the application to select the user object's type during execution	Syntax 2

Syntax 1

For user objects of a known data type

Description

Opens a custom visual user object of a known data type as a tab page in a Tab control.

Applies to

Tab controls

Syntax

tabcontrolname.**OpenTab** (*userobjectvar*, *index*)

Argument	Description
tabcontrolname	The name of the Tab control in which you want to open the user object as a tab page
userobjectvar	The name of the custom visual user object you want to open as a tab page. You can specify a custom visual 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 custom visual user object in *userobjectvar*
index	The number of the tab before which you want to insert the new tab. If *index* is 0 or greater than the number of tabs, the tab page is inserted at the end

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, OpenTab returns NULL.

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.

The tab page for the user object does not become selected. Scripts for constructor events of the controls on the user object do not run until the tab page is selected.

You must open a user object before you can access the properties of the user object. If you access the user object's properties before you open it, an execution error will occur.

A user object that is part of a Tab control's definition (that is, it was added to the Tab control in the Window painter) does not have to be opened in a script. PowerBuilder opens it when it opens the window containing the Tab control.

OpenTab *will* add the newly opened user object to the Tab control's Control array, which is a property that lists the tab pages within the Tab control.

Opening the same object twice
If you call Syntax 1 twice to open the same user object, PowerBuilder does open the user object again as another tab page, in contrast to the behavior of Open and OpenUserObject.

Examples

This statement opens an instance of a user object named u_Employee as a tab page in the Tab control tab_1:

```
tab_1.OpenTab(u_Employee, 0)
```

The following statements open an instance of a user object u_to_open as a tab page in the Tab control tab_1. It becomes the first tab in the control:

```
u_employee u_to_open
tab_1.OpenTab(u_to_open, 1)
```

See also

OpenTabWithParm

Syntax 2 For user objects of unknown data type

Description

Opens a visual user object as a tab page within a Tab control when the data type of the user object is not known until the script is executed.

Applies to

Tab controls

Syntax

tabcontrolname.**OpenTab** (*userobjectvar*, *userobjecttype*, *index*)

Argument	Description
tabcontrolname	The name of the Tab control in which you want to open the user object as a tab page
userobjectvar	A variable of data type DragObject. OpenTab 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 open. The data type of *userobjecttype* must be a descendant of *userobjectvar*
index	The number of the tab before which you want to insert the new tab. If *index* is 0 or greater than the number of tabs, the tab page is inserted at the end

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, OpenTab returns NULL.

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.

The tab page for the user object does not become selected. Scripts for Constructor events of the controls on the user object do not run until the tab page is selected.

You must open a user object before you can access the properties of the user object. If you access the user object's properties before you open it, an execution error will occur.

A user object that is part of a Tab control's definition (that is, it was added to the Tab control in the Window painter) does not have to be opened in a script. PowerBuilder opens it when it opens the window containing the Tab control.

OpenTab *will* add the newly opened user object to the Tab control's Control array, which is a property that lists the tab pages within the Tab control.

Considerations when specifying a user object type

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 *userobjectvar*. To refer to the instance in scripts, use *userobjectvar*.

If *userobjecttype* is a descendent user object, you can only refer to properties, events, functions, or structures that are part of the definition of *userobjectvar*. 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

The following example opens a user object as the last tab page in the Tab control tab_1. The user object is of the type specified in the string s_u_name and stores the reference to the user object in the variable u_to_open:

```
DragObject u_to_open
string s_u_name

s_u_name = sle_user.Text
tab_1.OpenTab(u_to_open, s_u_name, 0)
```

See also

OpenTabWithParm

OpenTabWithParm

Adds a visual user object to the specified window and makes all its properties and controls available to scripts, as OpenTab does. OpenTabWithParm also stores a parameter in the system's Message object so that it is accessible to the opened object.

To open	Use
A user object as a tab page	Syntax 1
A user object as a tab page, allowing the application to select the user object's type during execution	Syntax 2

Syntax 1

For user objects of a known data type

Description

Opens a custom visual user object of a known data type as a tab page in a Tab control and stores a parameter in the system's Message object.

Applies to

Tab controls

Syntax

tabcontrolname.**OpenTabWithParm** (*userobjectvar*, *parameter*, *index*)

Argument	Description
tabcontrolname	The name of the Tab control in which you want to open the user object as a tab page
userobjectvar	The name of the custom visual user object you want to open as a tab page. You can specify a custom visual 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. OpenTab places a reference to the opened custom visual 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
index	The number of the tab before which you want to insert the new tab. If *index* is 0 or greater than the number of tabs, the tab page is inserted at the end

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, OpenTabWithParm returns NULL.

Usage

The system Message object has three properties for storing data. Depending on the data type of the parameter specified for OpenTabWithParm, scripts for the opened user object would check one of the following properties.

Message object property	Argument 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, which causes an error. 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.

FOR INFO See also the usage notes for OpenTab, all of which apply to OpenTabWithParm.

Examples

This statement opens an instance of a user object named u_Employee as a tab page in the Tab control tab_empsettings. It also 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 tab page has the following statement:

```
tab_empsettings.OpenTabWithParm(u_Employee, &
"James Newton", 0)
```

The user object's Constructor event script has the following statement:

```
st_empname.Text = Message.StringParm
```

The following statements open an instance of a user object u_to_open as the first tab page in the Tab control tab_empsettings and store a number in message.DoubleParm. The last statement selects the tab page:

```
u_employee u_to_open
integer age = 50
tab_1.OpenTabWithParm(u_to_open, age, 1)
tab_1.SelectTab(u_to_open)
```

See also

OpenTab

Syntax 2

For user objects of unknown data type

Description

Opens a visual user object as a tab page within a Tab control when the data type of the user object is not known until the script is executed. In addition, OpenTabWithParm stores a parameter in the system's Message object so that it is accessible to the opened object.

Applies to

Tab controls

Syntax

tabcontrolname.**OpenTabWithParm** (*userobjectvar*, *parameter*, *userobjecttype*, *index*)

Argument	Description
tabcontrolname	The name of the Tab control in which you want to open the user object as a tab page
userobjectvar	A variable of data type DragObject. OpenTab 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*
index	The number of the tab before which you want to insert the new tab. If *index* is 0 or greater than the number of tabs, the tab page is inserted at the end

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, OpenTabWithParm returns NULL.

Usage

The system Message object has three properties for storing data. Depending on the data type of the parameter specified for OpenTabWithParm, scripts for the opened user object would check one of the following properties.

Message object property	Argument 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, which causes an error. 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.

FOR INFO See also the usage notes for OpenTab, all of which apply to OpenTabWithParm.

Examples

The following statement opens an instance of a user object u_data of type u_benefit_plan as the last tab page in the Tab control tab_1. The parameter "Benefits" is stored in message.StringParm:

```
DragObject u_data
tab_1.OpenTabWithParm(u_data, &
"Benefits", "u_benefit_plan", 0)
```

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 becomes the third tab page in the Tab control tab_1:

```
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 ...  ;

tab_1.OpenTabWithParm(u_to_open, &
e_location, s_u_name, 3)
```

The following 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 parameter is numeric so it is stored in message.DoubleParm. The user object becomes the first tab page in the Tab control tab_1:

```
userobject u_to_open
integer age = 60
string s_u_name

s_u_name = sle_user.Text
tab_1.OpenTabWithParm(u_to_open, age, &
s_u_name, 1)
```

See also

OpenTab

OpenUserObject

Adds a user object to the specified window and makes all its properties and controls available to scripts.

To	Use
Open an instance of a particular user object	Syntax 1
Open a user object, allowing the application to select the user object's type during execution	Syntax 2

Syntax 1 For user objects of a known data type

Description

Opens a user object of a known data type.

Applies to

Window objects

Syntax

windowname.**OpenUserObject** (*userobjectvar* {, *x*, *y* })

Argument	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, OpenUserObject returns NULL.

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 properties of the user object. If you access the user object's properties before you open it, an execution error will occur.

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.

OpenUserObject *will* add the newly opened user object to the window's Control array, which is a property that lists the window's controls.

When you open a user object during execution, the window does not destroy the user object automatically when you close the window. You need to call CloseUserObject to destroy the user object, usually when the window closes. If you don't destroy the user object, it holds on to its allocated memory, resulting in a memory leak.

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 property of a control).

Calling OpenUserObject twice
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.

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)
```

See also

OpenUserObjectWithParm

Syntax 2 For user objects of unknown data type

Description

Opens a user object when the data type of the user object is not known until the script is executed.

Applies to

Window objects

Syntax

windowname.**OpenUserObject** (*userobjectvar*, *userobjecttype* {, *x*, *y* })

Argument	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
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. If any argument's value is NULL, OpenUserObject returns NULL.

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 properties of the user object. If you access the user object's properties before you open it, an execution error will occur.

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.

OpenUserObject *will* add the newly opened user object to the window's Control array, which is a property that lists the window's controls.

When you open a user object during execution, the window does not destroy the user object automatically when you close the window. You need to call CloseUserObject to destroy the user object, usually when the window closes. If you don't destroy the user object, it holds on to its allocated memory, resulting in a memory leak.

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 property of a control).

The userobjecttype argument

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 *userobjectvar*. To refer to the instance in scripts, use *userobjectvar*.

If *userobjecttype* is a descendent user object, you can only refer to properties, events, functions, or structures that are part of the definition of *userobjectvar*. 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 your application.

Examples

The following example displays a user object of the type specified in the string s_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

s_u_name = sle_user.Text
w_info.OpenUserObject(u_to_open, s_u_name, 100, 200)
```

See also

OpenUserObjectWithParm

OpenUserObjectWithParm

Adds a user object to the specified window and makes all its properties 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.

To	Use
Open an instance of a particular user object	Syntax 1
Open a user object, allowing the application to select the user object's type during execution	Syntax 2

Syntax 1

For user objects of a known data type

Description

Opens a user object of a known data type and stores a parameter in the system's Message object.

Applies to

Window objects

Syntax

windowname.**OpenUserObjectWithParm** (*userobjectvar*, *parameter* {, *x*, *y* })

Argument	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

Argument	Description
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. If any argument's value is NULL, OpenUserObjectWithParm returns NULL.

Usage

The system Message object has three properties 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 properties.

Message object property	Argument 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, which causes an error. 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.

FOR INFO 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)
```

See also

CloseWithReturn
OpenUserObject
OpenWithParm

Syntax 2

For user objects of unknown data type

Description

Opens a user object when the data type of the user object is not known until the script is executed. In addition, OpenUserObjectWithParm stores a parameter in the system's Message object so that it is accessible to the opened object.

Applies to

Window objects

Syntax

windowname.**OpenUserObjectWithParm** (*userobjectvar, parameter, userobjecttype* {, *x, y* })

Argument	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

Argument	Description
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. If any argument's value is NULL, OpenUserObjectWithParm returns NULL.

Usage

The system Message object has three properties 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 properties.

Message object property	Argument 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, which causes an error. 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.

FOR INFO See also the usage notes for OpenUserObject, all of which apply to OpenUserObjectWithParm.

Examples

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 s_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

s_u_name = sle_user.Text
w_emp.OpenUserObjectWithParm(u_to_open, age, &
s_u_name, 100, 200)
```

See also

CloseWithReturn
OpenUserObject
OpenWithParm

OpenWithParm

Displays a window and makes all its properties 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.

To	Use
Open an instance of a particular window data type	Syntax 1
Allow the application to select the window's data type when the script is executed	Syntax 2

Syntax 1

For windows of a known data type

Description

Opens a window object of a known data type. OpenWithParm displays the window and makes all its properties and controls available to scripts. It also stores a parameter in the system's Message object.

Applies to

Window objects

Syntax

OpenWithParm (*windowvar, parameter* {, *parent* })

Argument	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) (optional)	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, OpenWithParm returns NULL.

Usage

The system Message object has three properties 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 properties.

Message object property	Argument 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, which causes an error. 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 property 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.

FOR INFO 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)
```

See also

CloseWithReturn
Open

Syntax 2

For windows of unknown data type

Description

Opens a window object when you don't know its data type until the application is running. OpenWithParm displays the window and makes all its properties and controls available to scripts. It also stores a parameter in the system's Message object.

Applies to

Window objects

Syntax

OpenWithParm (*windowvar*, *parameter*, *windowtype* {, *parent* })

Argument	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, OpenWithParm returns NULL.

Usage

The system Message object has three properties 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 properties.

Message object property	Argument 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, which causes an error. 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 property 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.

FOR INFO See also the usage notes for Open, all of which apply to OpenWithParm.

Examples

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_emp1"

OpenWithParm(wc_to_open, age, c_w_name, w_emp)
```

See also

CloseWithReturn
Open

OutgoingCallList

Description

Provides a list of the calls to other routines included in a performance analysis model.

Applies to

ProfileLine and ProfileRoutine objects

Syntax

instancename.**OutgoingCallList** (*list, aggregateduplicateroutinecalls*)

Argument	Description
instancename	Instance name of the ProfileLine or ProfileRoutine object
list	An unbounded array variable of data type ProfileCall in which OutgoingCallList stores a ProfileCall object for each call to other routines from within this routine. This argument is passed by reference
aggregateduplicateroutinecalls	A boolean indicating whether duplicate routine calls will result in the creation of a single or of multiple ProfileCall objects

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- ModelNotExistsError!—The model does not exist

Usage

You use the OutgoingCallList function to extract a list of the calls from a line and/or routine to other routines in a performance analysis model. You must have previously created the performance analysis model from a trace file using the BuildModel function. Each caller is defined as a ProfileCall object and provides the called routine and the calling routine, the number of times the call was made, and the elapsed time. The routines are listed in no particular order.

The *aggregateduplicateroutinecalls* argument indicates whether duplicate routine calls will result in the creation of a single or of multiple ProfileCall objects. This argument has no effect unless line tracing is enabled and a calling routine calls the current routine from more than one line. If *aggregateduplicateroutinecalls* is TRUE, a new ProfileCall object is created that aggregates all calls from the calling routine to the current routine. If *aggregateduplicateroutinecalls* is FALSE, multiple ProfileCall objects are returned, one for each line from which the calling routine called the called routine.

Examples

This example gets a list of the routines included in a performance analysis model and then gets a list of the routines called by each routine:

```
Long ll_cnt
ProfileCall lproc_call[]

lpro_model.BuildModel()
lpro_model.RoutineList(iprort_list)

FOR ll_cnt = 1 TO UpperBound(iprort_list)
   iprort_list[ll_cnt].OutgoingCallList(lproc_call, &
   TRUE)
   ...
NEXT
```

See also

BuildModel
IncomingCallList

PageCount

Description

Returns the total number of pages in the document in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**PageCount** ( )

Argument	Description
rtename	The name of the RichTextEdit control in which you want the page count

Return value

Integer. Returns the number of pages in the RichTextEdit control. Returns 1 if the control contains no text and -1 if an error occurs.

Usage

The number of pages in the document is determined by the amount of text and the layout specifications, such as page size, margins, font size, and so on.

When the RichTextEdit control shares data with a DataWindow, there is an instance of the document for each row of the DataWindow. PageCount reports the page count of a single instance. Multiply the value of the DataWindow's RowCount function by the page count to get the total number of pages.

Examples

This example displays the number of pages in the document in the RichTextEdit rte_1 as the text of the StaticText st_status:

```
st_status.Text = String(rte_1.PageCount())
```

See also

LineCount
LineLength
RowCount

PageCreated

Description

Reports whether a tab page has been created.

Applies to

User objects used as tab pages

Syntax

userobject.**PageCreated** ( )

Argument	Description
userobject	The name of the tab page whose existence you want to test

Return value

Boolean. Returns TRUE if the user object is a tab page and has been created and FALSE if the user object is not a tab page or has not been created.

Usage

A window will open more quickly if the creation of graphical representations is delayed for tab pages with many controls. However, scripts cannot refer to a control on a tab page until the tab page's Constructor event has run and a graphical representation of the control has been created. When the CreateOnDemand property of the Tab control is selected, scripts cannot reference controls on tab pages that the user hasn't viewed. PageCreated allows you to test whether a particular tab page has already been created.

Examples

This example tests whether tabpage_2 has been created and, if not, creates it:

```
IF tab_1.CreateOnDemand = True THEN
   IF tab_1.tabpage_2.PageCreated() = False THEN
      tab_1.tabpage_2.CreatePage()
   END IF
END IF
```

See also

CreatePage

ParentWindow

Description

Obtains the parent window of a window.

Applies to

Window objects

Syntax

windowname.**ParentWindow** ( )

Argument	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 or if *windowname* is NULL.

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 but when used in an MDI application, the parent of the popup is the MDI frame. A popup window opened from the application's Open event does not have a parent.

The ParentWindow property of the Menu object can be used like a pronoun in Menu scripts. It identifies the window that the menu is associated with when your program is running. For more information, see the *PowerBuilder User's Guide*.

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 controls, the OLE object on the clipboard replaces any object already in the control.

Platform information
When applied to OLE controls, this and other OLE functions have no effect on Macintosh and UNIX.

Applies to

EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, DropDownPictureListBox, DataWindow, OLE controls

Syntax

controlname.**Paste** ( )

Argument	Description
controlname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, DropDownPictureListBox, or OLE control into which you want to insert the contents of the clipboard. If *controlname* is a DataWindow, text is pasted into the edit control over the current row and column. If *controlname* is a DropDownListBox or DropDownPictureListBox, the AllowEdit property must be TRUE

Return value

Long. If *controlname* is NULL, Paste returns NULL.

For edit controls, returns the number of characters that were pasted into *controlname*. If nothing has been cut or copied (the clipboard is empty), the Paste function does not change the contents of the edit control and returns 0. If the clipboard contains nontext data (for example, a bitmap or OLE object) and the control cannot accept that data, Paste does not change the contents and returns 0.

For OLE 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.

You can paste bitmaps, as well as text, into a RichTextEdit control.

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 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.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE controls

Syntax

olecontrol.**PasteLink** ( )

Argument	Description
olecontrol	The name of the OLE 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

If *ole2control* is NULL, PasteLink returns NULL.

Usage

When you copy data to the clipboard from an application that supports OLE (the server application), you can paste the object into PowerBuilder's OLE 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

PasteRTF

Description

Pastes rich text data from a string into a DataWindow control, DataStore object, or RichTextEdit control.

Applies to

DataWindow controls, DataStore objects, and RichTextEdit controls

Syntax

rtename.**PasteRTF** (*richtextstring*, { *band* })

Argument	Description
rtename	The name of the DataWindow control, DataStore object, or RichTextEdit control into which you want to paste data in rich text format. The DataWindow object in the DataWindow control or DataStore must be a RichTextEdit DataWindow
richtextstring	A string whose value is data with rich text formatting
band (optional)	A value of the Band enumerated data type specifying the band into which the rich text data is pasted. Values are: ♦ Detail! — The data is pasted into the detail band ♦ Header! — The data is pasted into the header band ♦ Footer! — The data is pasted into the footer band The default is the band that contains the insertion point

Return value

Long. Returns the number of characters pasted if it succeeds and -1 if an error occurs. If *richtextstring* is NULL, PasteRTF returns NULL.

Usage

A DataWindow in the RTE presentation style has only three bands. There are no summary or trailer bands and there are no group headers and footers.

Examples

This statement pastes rich text in the string ls_richtext into the header of the RichTextEdit rte_message:

```
string ls_richtext
rte_message.PasteRTF(ls_richtext, Header!)
```

See also

CopyRTF

PasteSpecial

Description

Displays a standard OLE 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.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE controls

Syntax

olecontrol.**PasteSpecial** ( )

Argument	Description
olecontrol	The name of the OLE 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

If *ole2control* is NULL, PasteSpecial returns NULL.

Usage

For information about when an object on the clipboard is linkable, see PasteLink.

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*)

Argument	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. If *n* is NULL, Pi returns NULL.

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 the *DataWindow Reference*

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*)

Argument	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. If any argument's value is NULL, PixelsToUnits returns NULL.

Examples

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 ( )**

Argument	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. If *objectname* is NULL, PointerX returns NULL.

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** ( )

Argument	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. If *objectname* is NULL, PointerY returns NULL.

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

Menu objects

Syntax

menuname.**PopMenu** (*xlocation*, *ylocation*)

Argument	Description
menuname	The fully qualified name of a menu 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. If any argument's value is NULL, PopMenu returns NULL.

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 property of the menu is FALSE, you must make the menu 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.)

Calling PopMenu in an object script
PopMenu must be called in an object script. It should not be called in a global function.

Examples

These statements display the menu 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 menu 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 includes a menu 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())
```

PopulateError

Description

Fills in the Error object without causing a SystemError event.

Syntax

PopulateError (*number, text*)

Argument	Description
number	The integer to be stored in the number property of the Error object
text	The string to be stored in text property of the Error object

Return value

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

Usage

If the values you want to populate the Error object with depend on the current value of a variable in your script, you can use PopulateError to assign values to the number and text fields in the Error object (the remaining fields of the Error object will be populated automatically, including the line number of the error). Then you can call SignalError without arguments to trigger a SystemError. 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.

Examples

The following statements set the number and text values in the Error object according to a script variable, then trigger a SystemError event once the processing is complete:

```
// This function takes a table name and a record
// number, calls a routine gf_DoSomething to do
// something with the customer record id.
// The routine returns 0 on success or a negative
// number on error.

li_result = gf_DoSomething("Company", record_id)

IF (li_result < 0) THEN
   CHOOSE CASE li_result
   CASE -1
      PopulateError(1, "No company record exists &
         record id: " + record_id)
   CASE -2
      PopulateError(2, "That company record is &
         currently locked. Please try again later.")
```

```
        CASE -3
            PopulateError(3, "The company record could &
                not be updated.")
        CASE else
            PopulateError(999, "Update failed.")
        END CHOOSE
        SignalError()
    END IF
```

See also SignalError

Pos

Description

Finds one string within another string.

Syntax

Pos (*string1*, *string2* {, *start* })

Argument	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. If any argument's value is NULL, Pos returns NULL.

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

GetValue
GetObjectAtPointer
Left
Mid
Right
Pos in the *DataWindow Reference*

Position

Reports the position of the insertion point in an editable control.

To report	Use
The position of the insertion point in any editable control (except RichTextEdit)	Syntax 1
The position of the insertion point or the start and end of selected text in a RichTextEdit control or a DataWindow whose object has the RichTextEdit presentation style	Syntax 2

Syntax 1 For editable controls, except RichTextEdit

Description

Determines the position of the insertion point in an edit control.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, or DropDownListBox, DropDownPictureListBox controls

Syntax

editname.**Position** ( )

Argument	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, or DropDownListBox, or DropDownPictureListBox control in which you want to find the location of the insertion point

Return value

Long. Returns the location of the insertion point in *editname* if it succeeds and -1 if an error occurs. If *editname* is NULL, Position returns NULL.

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

Syntax 2

For RichTextEdit controls

Description

Determines the line and column position of the insertion point or the start and end of selected text in an RichTextEdit control.

Applies to

RichTextEdit and DataWindow controls

Syntax

rtename.**Position** (*fromline*, *fromchar* {, *toline*, *tochar* })

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control in which you want to find the location of the insertion point or selected text. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow
fromline	A long variable in which you want to save the number of the line where the insertion point or the start of the selection is
fromchar	A long variable in which you want to save the number in the line of the first character in the selection or after the insertion point
toline (optional)	A long variable in which you want to save the number of the line where the selection ends
tochar (optional)	A long variable in which you want to save the number in the line of the character before which the selection ends

Return value

Band enumerated data type. Returns the band (Detail!, Header!, or Footer!) containing the selection or insertion point.

Usage

Position reports the position of the insertion point if you omit the *toline* and *tochar* arguments. If text is selected, the insertion point can be at the beginning or the end of the selection. For example, if the user dragged down to select text, the insertion point is at the end.

If there is a selection, a character argument can be set to 0 to indicate that the selection begins or ends at the start of a line, with nothing else selected on that line. When the user drags up, the selection can begin at the start of a line and *fromchar* is set to 0. When the user drags down, the selection can end at the beginning of a line and *tochar* is set to 0.

Selection or insertion point To find out whether there is a selection or just an insertion point, specify all four arguments. If *toline* and *tochar* are set to 0, then there is no selection, only an insertion point. If there is a selection and you want the position of the insertion point, you will have to call Position again with only two arguments. This difference is described next.

The position of the insertion point and end of selection can differ When reporting the position of selected text, the positions are inclusive—Position reports the first line and character and the last line and character that are selected. When reporting the position of the insertion point, Position identifies the character just after the insertion point. Therefore, if text is selected and the insertion point is at the end, the values for the insertion point and the end of the selection differ.

To illustrate, suppose the first four characters in line 1 are selected and the insertion point is at the end. If you request the position of the insertion point:

```
rte_1.Position(ll_line, ll_char)
```

Then:

- ll_line is set to 1
- ll_char is set to 5, the character following the insertion point

If you request the position of the selection:

```
rte_1.Position(ll_startline, ll_startchar, &
ll_endline, ll_endchar)
```

- ll_startline and ll_startchar are both set to 1
- ll_endline is 1 and ll_endchar is set to 4, the last character in the selection

Passing values to SelectText Because values obtained with Position provide more information that simply a selection range, you cannot pass the values directly to SelectText. In particular, 0 is not a valid character position when selecting text, although it is meaningful in describing the selection.

Examples

This example calls Position to get the band and the line and column values for the beginning and end of the selection. The values are converted to strings and displayed in the StaticText st_status:

```
integer li_rtn
long ll_startline, ll_startchar
long ll_endline, ll_endchar
string ls_s, ls_band
band l_band

// Get the band and start and end of the selection
l_band = rte_1.Position(ll_startline, ll_startchar,&
ll_endline, ll_endchar)

// Convert position values to strings
ls_s = "Start line/char: " + String(ll_startline) &
+ ", " + String(ll_startchar)
ls_s = ls_s + " End line/char: " &
+ String(ll_endline) + ", " + String(ll_endchar)

// Convert Band data type to string
CHOOSE CASE l_band
CASE Detail!
ls_band = " Detail"
CASE Header!
ls_band = " Header"
CASE Footer!
ls_band = " Footer"
CASE ELSE
ls_band = " No band"
END CHOOSE
ls_s = ls_s + ls_band

// Display the information
st_status.Text = ls_s
```

This example extends the current selection down 1 line. It takes into account whether there is an insertion point or a selection, whether the insertion point is at the beginning or end of the selection, and whether the selection ends at the beginning of a line:

```
integer rtn
long l1, c1, l2, c2, linsert, cinsert
long l1select, c1select, l2select, c2select

// Get selectio start and end
rte_1.Position(l1, c1, l2, c2)
// Get insertion point
```

```
rte_1.Position(linsert, cinsert)

IF l2 = 0 and c2 = 0 THEN //insertion point
l1select = linsert
c1select = cinsert
l2select = l1select + 1 // Add 1 to end line
c2select = c1select

ELSEIF l2 > l1 THEN // Selection, ins pt at end
IF c2 = 0 THEN // End of selection (ins pt)
// at beginning of a line (char 0)
c2 = 999 // Change to end of prev line
l2 = l2 - 1
END IF

l1select = l1
c1select = c1
l2select = l2 + 1 // Add 1 to end line
c2select = c2

ELSEIF l2 < l1 THEN // selection, ins pt at start
IF c1 = 0 THEN // End of selection (not ins pt)
// at beginning of a line
c1 = 999 // Change to end of prev line
l1 = l1 - 1
END IF
l1select = l2
c1select = c2
l2select = l1 + 1 // Add 1 to end line
// (start of selection)
c2select = c1

ELSE // l1 = l2, selection on one line
l1select = l1
l2select = l2 + 1 // Add 1 to line
IF c1 < c2 THEN // ins pt at end
c1select = c1
c2select = c2
ELSE // c1 > c2, ins pt at start
c1select = c2
c2select = c1
END IF
END IF
```

```
// Select the extended selection
rtn = rte_1.SelectText( l1select, c1select, &
l2select, c2select )
```

For an example of selecting each word in a RichTextEdit control, see SelectTextWord.

See also

SelectedLine
SelectedStart
SelectText

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*)

Argument	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. If any argument's value is NULL, Post returns NULL.

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.

Examples

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* })

Argument	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 property 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 property of the system's Message object. When you specify a string, a pointer to the string is stored in the LongParm property, 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*. If any argument's value is NULL, PostEvent returns NULL.

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 menu delete data, the button's Clicked script can perform the deletion and the menu'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, 0)
```

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

PostURL

Description

Performs an HTTP Post, allowing a PowerBuilder application to send a request through CGI, NSAPI, or ISAPI.

Applies to

Inet objects

Syntax

servicereference.**PostURL** (*urlname*, *urldata*, *headers*, *data*)

Argument	Description
servicereference	Reference to the Internet service instance
urlname	String specifying the URL to post
urldata	Blob specifying arguments to the URL specified by *urlname*
headers	String specifying HTML headers. In Netscape, a newline (~n) is required after each HTTP header and a final newline after all headers
data	InternetResult instance into which the function returns HTML

Return value

Integer. Returns values as follows:

1 Success
-1 General error
-2 Invalid URL
-4 Cannot connect to the Internet
-6 Internet request failed

Usage

Call this function to invoke a CGI, NSAPI, or ISAPI function.

Data references a standard class user object that descends from InternetResult and that has an overridden InternetData function. This overridden function then performs the processing you want with the returned HTML. Because the Internet returns data asynchronously, *data* must reference a variable that remains in scope after the function executes (such as a window-level instance variable).

FOR INFO For more information on the InternetResult standard class user object and the InternetData function, use the PowerBuilder Browser.

Examples

This example calls the PostURL function. Iinet_base is an instance variable of type inet:

```
Blob lblb_args
String ls_headers
String ls_url
Long ll_length

iir_msgbox = CREATE n_ir_msgbox
ls_url = "http://coltrane.sybase.com/"
ls_url += "cgi-bin/pbcgi60.exe/"
ls_url += "myapp/n_cst_html/f_test?"
lblb_args = blob("")
ll_length = Len(lblb_args)
ls_headers = "Content-Length: " &
   + String(ll_length) + "~n~n"
iinet_base.PostURL &
   (ls_url, lblb_args, ls_headers, iir_msgbox)
```

See also

GetURL
HyperLinkToURL
InternetData

Preview

Description

Displays the contents of a RichTextEdit control as either a preview of the document as it would print or in an editing view.

Applies to

RichTextEdit controls

Syntax

rtename.**Preview** (*previewsetting*)

Argument	Description
rtename	The name of the RichTextEdit control which you want to preview or edit
previewsetting	A boolean value indicating whether to put the RichTextEdit into preview or edit mode. Values are: ♦ True — Preview the contents of the RichTextEdit as it would look when printed ♦ False — Displays the contents in editable form

Return value

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

Usage

A RichTextEdit control has two ways of viewing the content: edit mode and preview mode. The Preview function switches between the two.

Edit mode Edit mode displays the text in readable form. The user can enter, select, and change text. There are properties for controlling the display of nonprinting characters in the text, such as carriage returns, spaces, tabs, and input fields. In edit mode, the toolbar, ruler bar, and tab bar, if visible, display above the editing area of the control.

Preview mode Preview mode displays a miniature page within the control. The page is sized to fit within the control. Preview mode provides edit boxes for specifying paper dimensions and margins. Any selection is canceled when the control switches to preview mode. The user cannot edit text in preview mode, but scripts can call functions for selecting and changing text, including inserting documents.

If you call ShowHeadFoot when the control is in preview mode, you return to edit mode with the header and footer editing panels displayed.

Size of RichTextEdit control
Make sure the RichTextEdit control is big enough to display the page formatting and scrolling controls available in preview mode.

Examples

This example previews the page layout of the RichTextEdit rte_1:

```
rte_1.Preview(TRUE)
```

This example displays the contents of the RichTextEdit rte_1 as editable text:

```
rte_1.Preview(FALSE)
```

See also

IsPreview

Print

Sends data to the current printer (or spooler, if the user has a spooler set up). There are several syntaxes.

To	Use
Send the contents of a DataWindow control or DataStore to the printer as a print job	Syntax 1
Include a visual object, such as a window or a graph control in a print job	Syntax 2
Send one or more lines of text as part of a print job	Syntax 3
Print the contents of an RTE control	Syntax 4

Syntax 1 For printing a single DataWindow or DataStore

Description

Sends the contents of a DataWindow control or DataStore object to the printer as a print job.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**Print** ({ *canceldialog* })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow that contains the information to be printed
canceldialog (optional)	A boolean value indicating whether you want to display a nonmodal dialog that allows the user to cancel printing. Values are: ♦ TRUE — (Default) Display the dialog ♦ FALSE — Do not display the dialog **Working with DataStore objects** When working with DataStores, the *canceldialog* argument must always be set to FALSE

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, Print returns NULL.

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. You don't need to use the PrintOpen and PrintClose functions.

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.

The printed output uses the same fonts and layout that appear on screen for the DataWindow object.

When the DataWindow object's presentation style is RichTextEdit, each row begins a new page in the printed output.

Events for DataWindow printing

When you use Print for DataWindow controls or DataStores, 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 return codes that let you control whether the page about to be formatted is printed. You can skip the upcoming page by returning a value of 1 in the PrintPage event.

Examples

This statement sends the contents of dw_employee to the current printer:

```
dw_employee.Print()
```

See also

PrintDataWindow

Syntax 2

For printing a visual object in a print job

Description

Includes a visual object, such as a window or a graph control, in a print job that you have started with the PrintOpen function.

Applies to

Any object

Syntax

objectname.**Print** (*printjobnumber, x, y* {, *width, height* })

Argument	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 page of the left corner of the object, 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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, Print returns NULL.

Usage

PowerBuilder manages print jobs by opening the job, sending data, and closing the job. When you use Syntax 2 or 3, you must call the PrintOpen function and the PrintClose or PrintCancel functions yourself to manage the process.

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.

Examples

This example prints the CommandButton cb_close in its original size at location 500, 1000:

```
long Job
Job = PrintOpen( )
cb_close.Print(Job, 500,1000)
PrintClose(Job)
```

This example opens a print job, which defines a new page, then prints a title using the third syntax of Print. Then it uses this syntax of Print to print a graph on the first page and a window on the second page:

```
long 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)
```

See also

PrintCancel
PrintClose
PrintOpen
PrintScreen

Syntax 3

For printing text in a print job

Description

Sends one or more lines of text as part of a print job that you have opened with the PrintOpen function. You can specify tab settings before or after the text. The tab settings control the text's horizontal position on the page.

Applies to

Not object-specific

Syntax

Print (*printjobnumber*, { *tab1*, } *string* {, *tab2* })

Argument	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, Print returns NULL.

Usage

PowerBuilder manages print jobs by opening the job, sending data, and closing the job. When you use Syntax 2 or 3, you must call the PrintOpen function and the PrintClose or PrintCancel functions yourself to manage the process.

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.

Examples

This example opens a print job, prints the string Sybase Corporation in the default font, and then starts a new line:

```
long Job

// Define a blank page and assign the job an ID
```

```
Job = PrintOpen( )

// Print the string and then start a new line
Print(Job, "Sybase Corporation")
...
PrintClose(Job)
```

This example opens a print job, prints the string Sybase Corporation in the default font, tabs 5 inches from the left edge of the print area but does not start a new line:

```
long 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, "Sybase Corporation", 5000)
...
PrintClose(Job)
```

The first Print statement below tabs half an inch from the left edge of the print area, prints the string Sybase 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:

```
long Job

// Define a blank page and assign the job an ID
Job = PrintOpen( )

// Print the string and start a new line
Print(Job, 500, "Sybase 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 Sybase 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:

```
long 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, "Sybase 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:

```
long 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

Syntax 4 For RichTextEdit controls

Description

Prints the contents of a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**Print** (*copies*, *pagerange*, *collate*, *canceldialog*)

Argument	Description
rtename	The name of the RichTextEdit control whose contents you want to print
copies	An integer specifying the number of copies you want to print.
pagerange	A string describing the pages you want to print. To print all pages, specify an empty string (""). To specify a subset of pages, use dashes to specify a range and commas to separate ranges and individual page numbers—for example, "1-3" or "2,5,8-10". When *rtename* shares data with a DataWindow, *pagerange* refers to pages based on the total number of pages in the control, not within each instance of the document
collate	A boolean value indicating whether you want the copies collated. Values are: ♦ TRUE — Collate copies ♦ FALSE — Do not collate copies
canceldialog	A boolean value indicating whether you want to display a nonmodal dialog that allows the user to cancel printing. Values are: ♦ TRUE — Display the dialog ♦ FALSE — Do not display the dialog

Return value

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

Usage

When the RichTextEdit control shares data with a DataWindow, the total number of pages contained in the control is the page count of the document multiplied by the row count of the DataWindow.

You can specify printed page numbers by including an input field in the header or footer of your document. By calling InputFieldChangeData in the PrintHeader or PrintFooter event, you can specify the current value of that input field.

Events Print may trigger these events:

- PrintHeader
- PrintFooter

Examples

This statement prints one copy of pages 1 to 5 of the document in the RichTextEdit control rte_1. The output is not collated and a dialog displays to allow the user to cancel the printing:

```
rte_1.Print(1, "1-5", FALSE, TRUE)
```

See also

Preview

PrintBitmap

Description

Writes a bitmap at the specified location on the current page.

Syntax

PrintBitmap (*printjobnumber, bitmap, x, y, width, height*)

Argument	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.
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. If any argument's value is NULL, PrintBitmap returns NULL.

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).

Examples

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:

```
long 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)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
PrintBitmap(Job, "HD:PB:Bitmap1.BMP", 50,100, 0,0)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
PrintBitmap(Job, "/export/home/pb/bitmap1.bmp", &
50,100, 0,0)
```

See also

PrintClose
PrintLine
PrintRect
PrintRoundRect
PrintOval
PrintOpen

PrintCancel

Cancels printing and deletes the spool file, if any. There are two syntaxes.

To	Use
Cancel printing of a DataWindow or DataStore printed with the Print function	Syntax 1
Cancel a print job that you began with the PrintOpen function	Syntax 2

Syntax 1

For DataWindows and DataStores

Description

Cancels the printing of a DataWindow or DataStore that was printed using Syntax 1 of Print.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**PrintCancel** ( )

Argument	Description
dwcontrol	The name of a DataWindow control, DataStore object, or child DataWindow that was printed with Syntax 1 of the Print function

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *dwcontrol* is NULL, PrintCancel returns NULL.

Usage

PrintCancel cancels the printing of the specified DataWindow or DataStore by deleting the spool file, if any, and closing the job.

When you use the Print function to print the DataWindow or DataStore, 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()
```

See also

Print

Syntax 2

For canceling a print job

Description

Cancels printing of a print job that you opened with the PrintOpen function. The print job is identified by the number returned by PrintOpen.

Syntax

PrintCancel (*printjobnumber*)

Argument	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. If *printjobnumber* is NULL, PrintCancel returns NULL.

Usage

PrintCancel cancels the specified print job 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.

Examples

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:

```
long 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)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
PrintBitmap(Job, "HD:PB:Bitmap1.BMP", 50,10, 0,0)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
PrintBitmap(Job, "/export/home/pb/bitmap1.bmp", &
50,10, 0,0)
```

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*)

Argument	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. If *printjobnumber* is NULL, PrintClose returns NULL.

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.

Examples

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:

```
ulong 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)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
PrintBitmap(Job, "HD:PB:Bitmap1.BMP", 50,10, 0,0)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
PrintBitmap(Job, "/export/home/pb/bitmap1.bmp", &
50,10, 0,0)
```

See also

PrintCancel
PrintOpen

PrintDataWindow

Description

Prints the contents of a DataWindow control as a print job.

Syntax

PrintDataWindow (*printjobnumber, dwcontrol*)

Argument	Description
printjobnumber	The number the PrintOpen function assigned to the print job
dwcontrol	The name of the DataWindow control or child DataWindow containing the DataWindow object you want to print

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, PrintDataWindow returns NULL.

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.

When the DataWindow's presentation style is RichTextEdit, each row begins a new page in the printed output.

FOR INFO For information on skipping individual pages with return codes in the PrintPage event, see the Print function.

Examples

These statements send the contents of three DataWindow controls to the current printer in a single print job:

```
long 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 properties. You can use the font number in the PrintSetFont or PrintText functions. You can define up to eight fonts at a time.

Syntax

PrintDefineFont (*printjobnumber, fontnumber, facename, height, weight, fontpitch, fontfamily, italic, underline*)

Argument	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 (not italic)
underline	A boolean value indicating whether the font is underlined. The default is FALSE (not underlined)

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, PrintDefineFont returns NULL.

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.

Examples

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:

```
long 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*)

Argument	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. If any argument's value is NULL, PrintLine returns NULL.

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 See also).

Examples

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:

```
long 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* })

Argument	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

Long. Returns the job number if it succeeds and -1 if an error occurs. If any argument's value is NULL, PrintOpen returns NULL.

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:

```
ulong li_job
li_job = PrintOpen()
```

This example opens a job and gives it a name:

```
ulong 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*)

Argument	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. If any argument's value is NULL, PrintOval returns NULL.

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).

Examples

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 four inches from the top and three inches from the left edge of the print area. Because its height and width are equal, the oval is actually a circle:

```
long 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*)

Argument	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. If any argument's value is NULL, PrintPage returns NULL.

Examples

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:

```
long Job

// Open a job with new blank page.
Job = PrintOpen()

// Print a bitmap on the page.
PrintBitmap(Job, "d:\PB\BITMAP1.BMP", 100,250, 0,0)

// Begin a new page.
PrintPage(Job)

// Print the company name on the new page.
Print(Job, "Sybase Corporation")
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
PrintBitmap(Job, "HD:PB:Bitmap1.BMP", 100,250, 0,0)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
PrintBitmap(Job, "/export/home/pb/bitmap1.bmp", &
100,250, 0,0)
```

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*)

Argument	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. If any argument's value is NULL, PrintRect returns NULL.

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).

Examples

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 four inches from the left and three inches from the top of the print area:

```
long 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*)

Argument	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. If any argument's value is NULL, PrintRoundRect returns NULL.

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).

Examples

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 one inch from the top and one 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:

```
long 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* })

Argument	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
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. If you specify *width*, you must also specify *height*
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. If any argument's value is NULL, PrintScreen returns NULL.

Examples

This statement prints the current screen image in its original size at location 500, 1000:

```
long 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* })

Argument	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
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. If any argument's value is NULL, PrintSend returns NULL.

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 zero, you can use another character for zero in *string* and specify the character that represents zero 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 zero.

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, and so on

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 zero:

```
long 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:

```
long 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 properties.

Syntax

PrintSetFont (*printjobnumber, fontnumber*)

Argument	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. If any argument's value is NULL, PrintSetFont returns NULL.

Examples

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:

```
long 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,"Sybase Corporation")
```

See also

PrintDefineFont
PrintOpen

PrintSetSpacing

Description

Sets the factor that PowerBuilder uses to calculate line spacing.

Syntax

PrintSetSpacing (*printjobnumber, spacingfactor*)

Argument	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. If any argument's value is NULL, PrintSetSpacing returns NULL.

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.

Examples

These statements start a new print job and set the vertical spacing factor to 1.5 (one and a half spacing):

```
long 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 lets the user specify settings for the printer.

Syntax

PrintSetup ( )

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, PrintSetup returns NULL.

Usage

For Windows 95 and Windows NT 3.51 and higher, the user's settings have effect for the duration of the application only. After the application exits, printer settings revert to their previous values.

For Windows 3.1, the user's settings become the new settings for the printer driver and affect subsequent print jobs for all applications.

Examples

These statements call the Printer Setup dialog box for the current system printer and then start a new print job:

```
long 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* })

Argument	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. If any argument's value is NULL, PrintText returns NULL.

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):

```
long 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:

```
long 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:

```
long 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*)

Argument	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 *rintjobnumber* if it succeeds and -1 if an error occurs. If any argument's value is NULL, PrintWidth returns NULL.

Examples

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:

```
long 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*)

Argument	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. If any argument's value is NULL, PrintX returns NULL.

Examples

These statements set LocX to the x coordinate of the cursor and print End of Report an inch beyond that location:

```
integer LocX
long 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*)

Argument	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. If any argument's value is NULL, PrintY returns NULL.

Examples

These statements print a bitmap one inch below the location of the print cursor:

```
integer LocX, LocY
long 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*)

Argument	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. If any argument's value is NULL, ProfileInt returns NULL.

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.

Windows NT and Windows 95

On 32-bit systems, ProfileInt can also be used to obtain configuration settings from the Windows system registry. For information on how to use the system registry, see the discussion of initialization files and the windows registry in *Application Techniques*.

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)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
ProfileInt("HD:System Folder:Preferences:My App:" + &
"Profile Preferences", "PB", "maximized", 3)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
ProfileInt("/export/home/myapp/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 the *DataWindow Reference*

ProfileString

Description

Obtains the string value of a setting in the profile file for your application.

Syntax

ProfileString (*filename, section, key, default*)

Argument	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 (""). If any argument's value is NULL, ProfileString returns NULL.

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.

Windows NT and Windows 95

On 32-bit systems, ProfileString can also be used to obtain configuration settings from the Windows system registry. For information on how to use the system registry, see the discussion of initialization files and the windows registry in *Application Techniques*.

Examples

These examples use a hypothetical file called PROFILE.INI, which contains the following lines. (Quotes around string values in the INI file are optional):

```
[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 Macintosh On Macintosh, the filename in the preceding code might look like this:

```
ProfileString("HD:System Folder:Preferences:My App:"&
+ "Profile Preferences", "Employee", "Name", "None")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
ProfileString("/export/home/myapp/profile.ini", &
"Employee", "Name", "None")
```

On the Macintosh, the following command finds the same information in the Profile Preferences file the PowerBuilder folder on the drive Hard Disk:

```
ProfileString("Hard Disk:PowerBuilder:Profile
Preferences",&
"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 the *DataWindow Reference*

Rand

Description

Obtains a random whole number between 1 and a specified upper limit.

Syntax

Rand (*n*)

Argument	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. If *n* is NULL, Rand returns NULL.

Usage

The sequence of numbers generated by repeated calls to the Rand function is a pseudorandom 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.

Examples

This statement returns a random whole number between 1 and 10:

```
Rand(10)
```

See also

Randomize
Rand in the *DataWindow Reference*

Randomize

Description

Initializes the random number generator so that the Rand function begins a new series of pseudorandom numbers.

Syntax

Randomize (*n*)

Argument	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. If *n* is NULL, Randomize returns NULL. The return value is never used.

Usage

The sequence of numbers generated by repeated calls to the Rand function is a computer-generated pseudorandom 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 pseudorandom 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

Reads data from an opened OLE stream object.

To	Use
Read data into a string	Syntax 1
Read data into a character array or blob	Syntax 2

Syntax 1 For reading into a string

Description

Reads data from an OLE stream object into a string.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLEStream objects

Syntax

olestream.**Read** (*variable* {, *stopforline* })

Argument	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

Integer. Returns the number of characters or bytes read. If an end-of-file mark (EOF) is encountered before any characters are read, Read returns -100. Read returns one of the following negative values if an error occurs:

-1 Stream is not open
-2 Read error
-9 Other error

If any argument's value is NULL, Read returns NULL.

Examples

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 string ls_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 ls_info

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

olestr_info.Open(stg_stuff, "info", &
stgRead!, stgExclusive!)
olestr_info.Read(ls_info)
```

See also

Open
Length
Seek
Write

Syntax 2

For character arrays or blobs

Description

Reads data from an OLE stream object into a character array or blob.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLEStream objects

Syntax

olestream.**Read** (*variable* {, *maximumread* })

Argument	Description
olestream	The name of an OLE stream variable that has been opened

Argument	Description
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

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

If any argument's value is NULL, Read returns NULL.

Examples

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*)

Argument	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. *Stringorblob* can also be an Any variable containing a string or blob

Return value

Real. Returns the value of *stringorblob* as a real. If *stringorblob* is not a valid PowerScript number or is an incompatible data type, Real returns 0. If *stringorblob* is NULL, Real returns NULL.

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, although 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 the *DataWindow Reference*

RegistryDelete

Description

Deletes a key or a value for a key in the Windows system registry.

Platform information
The registry functions have no effect on Macintosh and UNIX.

Syntax

RegistryDelete (*key, valuename*)

Argument	Description
key	A string whose value is the key in the system registry you want to delete or whose value you want to delete. To uniquely identify a key, specify the list of parent keys above it in the hierarchy, starting with the root key. The keys in the list are separated by backslashes
valuename	A string containing the name of a value in the registry. To delete the key, its named values, and all its subkeys, specify an empty string. When accessing the Windows 3.1 registry, which has no named values, you always specify an empty string for *valuename*

Return value

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

Usage

For more information about entries in the system registry, see RegistrySet.

Examples

On Windows NT, this statement deletes the value name Title and its associated value from the registry. The key is not deleted:

```
RegistryDelete( &
"HKEY_LOCAL_MACHINE\Software\MyApp.Settings\Fonts", &
"Title")
```

On Windows 3.1, this statement deletes the key Fonts from the registry:

```
RegistryDelete( &
"HKEY_CLASSES_ROOT\MyApp.Settings\Fonts", "")
```

See also

RegistryGet
RegistryKeys
RegistrySet
RegistryValues

RegistryGet

Description

Gets a value from the Windows system registry.

Platform information
The registry functions have no effect on Macintosh and UNIX.

Syntax

RegistryGet (*key, valuename, valuetype, valuevariable*)

Argument	Description
key	A string whose value names the key in the system registry whose value you want. To uniquely identify a key, specify the list of parent keys above it in the hierarchy, starting with the root key. The keys in the list are separated by backslashes
valuename	A string containing the name of a value in the registry. Each key can have one unnamed value and several named values. For the unnamed value, specify an empty string. When accessing the Windows 3.1 registry, which has no named values, you always specify an empty string for *valuename*
valuetype	A value of the RegistryValueType enumerated data type identifying the data type of a value in the registry. Values are: ♦ RegString!—A null-terminated string ♦ RegExpandString!—A null-terminated string that contains unexpanded references to environment variables ♦ RegBinary!—Binary data ♦ ReguLong!—A 32-bit number ♦ ReguLongBigEndian!—A 32-bit number ♦ RegLink!—A Unicode symbolic link ♦ RegMultiString!—An unbounded array of strings
valuevariable	A variable corresponding to the data type of *valuetype* in which you want to store the value obtained from the system registry for the specified key and value name

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. An error is returned if the data type of *valuevariable* does not correspond to the data type specified in *valuetype*.

Usage

Long string values (more than 2048 bytes) should be stored as files and the file name stored in the registry.

FOR INFO For more information about keys and value names in the system registry, see RegistrySet.

Examples

On Windows NT and Windows 95 This statement obtains the value for the name Title and stores it in the string ls_titlefont:

```
string ls_titlefont
RegistryGet( &
    "HKEY_LOCAL_MACHINE\Software\MyApp.Settings &
    \Fonts", "Title", RegString!, ls_titlefont)
```

This statement obtains the value for the name NameOfEntryNum and stores it in the long ul_num:

```
ulong ul_num
RegistryGet( &
    "HKEY_USERS\MyApp.Settings\Fonts", &
    "NameOfEntryNum", RegULong!, ul_num)
```

On Windows 3.1 This statement obtains the value for the key Fonts from the registry and stores it in the string ls_titlefont:

```
string ls_titlefont
RegistryGet( &
    "HKEY_CLASSES_ROOT\MyApp.Settings\Fonts", &
    "", RegString!, ls_titlefont)
```

See also

RegistryDelete
RegistryKeys
RegistrySet
RegistryValues

RegistryKeys

Description

Obtains a list of the keys that are child items (subkeys) one level below a key in the Windows system registry.

Platform information
The registry functions have no effect on Macintosh and UNIX.

Syntax

RegistryKeys (*key, subkeys*)

Argument	Description
key	A string whose value names the key in the system registry whose subkeys you want. To uniquely identify a key, specify the list of parent keys above it in the hierarchy, starting with the root key. The keys in the list are separated by backslashes
subkeys	An array variable of strings in which you want to store the subkeys. If the array is variable size, its upper bound will reflect the number of subkeys found. If the array is fixed size, it must be large enough to hold all the subkeys. However, there will be no way to know how many subkeys were actually found

Return value

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

Usage

For more information about entries in the system registry, see RegistrySet.

Examples

This example obtains the subkeys associated with the key HKEY_CLASSES_ROOT\MyApp. The subkeys are stored in the variable-size array ls_subkeylist:

```
string ls_subkeylist[]
integer li_rtn
li_rtn = RegistryKeys("HKEY_CLASSES_ROOT\MyApp", &
   ls_subkeylist)
IF li_rtn = -1 THEN
   ... // Error processing
END IF
```

See also

RegistryDelete
RegistryGet
RegistrySet
RegistryValues

RegistrySet

Description

Sets the value for a key and value name in the system registry. If the key or value name does not exist, RegistrySet creates a new key or name and sets its value.

Platform information
The registry functions have no effect on Macintosh and UNIX.

Syntax

RegistrySet (*key, valuename, valuetype, value*)

Argument	Description
key	A string whose value names the key in the system registry whose value you want to set. To uniquely identify a key, specify the list of parent keys above it in the hierarchy, starting with the root key. The keys in the list are separated by backslashes. If *key* does not exist in the registry, RegistrySet creates a new key. To create a *key* without a named value, specify an empty string for *valuename*
valuename	A string containing the name of a value in the registry. Each key may have several named values. To specify the unnamed value (the only option for Windows 3.1), specify an empty string. If *valuename* does not exist in the registry, RegistrySet causes a new name to be created for *key*
valuetype	A value of the RegistryValueType enumerated data type identifying the data type of a value in the registry. Values are: ♦ RegString!—A null-terminated string ♦ RegExpandString!—A null-terminated string that contains unexpanded references to environment variables ♦ RegBinary!—Binary data ♦ ReguLong!—A 32-bit number ♦ ReguLongBigEndian!—A 32-bit number ♦ RegLink!—A Unicode symbolic link ♦ RegMultiString!—An unbounded array of strings
value	A variable corresponding to the data type of *valuetype* containing a value to be set in the registry

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. An error is returned if the data type of *valuevariable* does not correspond to the data type specified in *valuetype*.

Usage

Long string values (more than 2048 bytes) should be stored as files and the file name stored in the registry.

The structure of the registry depends on the operating system, either 16-bit (Window 3.1) or 32-bit (Windows NT or Windows 95).

On 32-bit systems (Windows NT and Windows 95) The registry supports named values. Keys have the following structure.

Item	Description

Key	An element in the registry. A key is part of a tree of keys, descending from one of the predefined root keys. Each key is a subkey or child of the parent key above it in the hierarchy.
	There are four root strings:
	♦ HKEY_CLASSES_ROOT
	♦ HKEY_LOCAL_MACHINE
	♦ HKEY_USERS
	♦ HKEY_CURRENT_USER
	A key is uniquely identified by the list of parent keys above it. The keys in the list are separated by slashes, as shown in these examples.
	`HKEY_CLASSES_ROOT\Sybase.Application`
	`HKEY_USERS\MyApp\Display\Fonts`

Value name	The name of a value belonging to the key. A key can have one unnamed value and one or more named values
Value type	A value identifying the data type of a value in the registry
Value	A value associated with a value name or an unnamed value. Several string, numeric, and binary data types are supported by the registry

On 16-bit systems (Windows 3.1) Each entry in the system registry has two items. Windows 3.1 does not use value names.

Item	Description

Key	An element in the registry. A key is part of a tree of keys, descending from one of the predefined root keys. Each key is a subkey or child of the parent key above it in the hierarchy.
	HKEY_CLASSES_ROOT is the only root string in the Windows 3.1 registry.
	A key is uniquely identified by the list of parent keys above it. The keys in the list are separated by slashes, as shown in these examples.
	`HKEY_CLASSES_ROOT\Sybase.Application`
	`HKEY_CLASSES_ROOT\MyApp\Display\Fonts`

Value	The value for the key. The Windows 3.1 registry has string values only

Examples

On Windows NT and Windows 95 This example sets a value for the key Fonts and the value name Title:

```
RegistrySet( &
    "HKEY_LOCAL_MACHINE\Software\MyApp\Fonts", &
    "Title", RegString!, sle_font.Text)
```

This statement sets a value for the key Fonts and the value name NameOfEntryNum:

```
ulong ul_num
RegistrySet( &
    "HKEY_USERS\MyApp.Settings\Fonts", &
    "NameOfEntryNum", RegULong!, ul_num)
```

On Windows 3.1 This example sets a value for the key Title, a subkey of Fonts:

```
RegistrySet( &
    "HKEY_CLASSES_ROOT\MyApp\Fonts\Title", &
    "", RegString!, sle_font.Text)
```

See also

RegistryDelete
RegistryGet
RegistryKeys
RegistryValues

RegistryValues

Description

Obtains the list of named values associated with a key.

Platform information
The registry functions have no effect on Macintosh and UNIX.

Syntax

RegistryValues (*key, valuename*)

Argument	Description
key	A string whose value is the key in the system registry for which you want the values of its subkeys. To uniquely identify a key, specify the list of parent keys above it in the hierarchy, starting with the root key. The keys in the list are separated by backslashes
valuename	An array variable of strings in which you want to store the names. If the array is variable size, its upper bound will reflect the number of named values found. If the array is fixed size, it must be large enough to hold all the names. However, there will be no way to know how many names were actually found

Return value

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

Usage

For more information about entries in the system registry, see RegistrySet.

Examples

On Windows NT, this example gets the value names associated with the key Fonts and stores them in the array ls_valuearray:

```
string ls_valuearray[]
RegistryValues( &
"HKEY_LOCAL_MACHINE\Software\MyApp.Settings\Fonts",&
ls_valuearray)
```

See also

RegistryDelete
RegistryGet
RegistryKeys
RegistrySet

RelativeDate

Description

Obtains the date that occurs a specified number of days after or before another date.

Syntax

RelativeDate (*date*, *n*)

Argument	Description
date	A value of type date
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. If any argument's value is NULL, RelativeDate returns NULL.

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 the *DataWindow Reference*

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*)

Argument	Description
time	A value of type time
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. If any argument's value is NULL, RelativeTime returns NULL.

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 the *DataWindow Reference*

ReleaseAutomationNativePointer

Description

Releases the pointer to an OLE object that you got with GetAutomationNativePointer.

Applies to

OLEObject

Syntax

oleobject.**ReleaseAutomationNativePointer** (*pointer*)

Argument	Description
oleobject	The name of an OLEObject variable containing the object for which you want to release the native pointer
pointer	A UnsignedLong variable that holds the pointer you want to release. ReleaseAutomationNativePointer sets *pointer* to 0 so that it is clearly no longer a valid pointer

Return value

Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage

Using the pointer in your own DLL calls *Pointer* is a pointer to OLE's IUnknown interface. You can use it with QueryInterface() to get other interface pointers.

When you call GetAutomationNativePointer, PowerBuilder calls OLE's AddRef() function, which locks the pointer. You can release the pointer in your DLL function or in a PowerBuilder script with the ReleaseAutomationNativePointer function.

Examples

See GetAutomationNativePointer.

See also

GetAutomationNativePointer
GetNativePointer
ReleaseNativePointer

ReleaseNativePointer

Description

Releases the pointer to an OLE object that you got with GetNativePointer.

Applies to

OLE controls and OLE custom controls

Syntax

olename.**ReleaseNativePointer** (*pointer*)

Argument	Description
olename	The name of the OLE control containing the object for which you want the native pointer
pointer	A UnsignedLong variable that holds the pointer you want to release. ReleaseNativePointer sets *pointer* to 0 so that it is clearly no longer a valid pointer

Return value

Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage

Using the pointer in your own DLL calls *Pointer* is a pointer to OLE's IUnknown interface. You can use it with QueryInterface() to get other interface pointers.

When you call GetNativePointer, PowerBuilder calls OLE's AddRef( ) function, which locks the pointer. You can release the pointer in your DLL function or in a PowerBuilder script with the ReleaseNativePointer function.

Examples

See GetNativePointer.

See also

GetAutomationNativePointer
GetNativePointer
ReleaseAutomationNativePointer

RemoteStopConnection

Description

Allows a client application to disconnect another client from a server application.

This function applies to distributed applications only.

Applies to

Connection objects

Syntax

connection.**RemoteStopConnection** (*disconnectid*)

Argument	Description
connection	The name of the Connection object originally used to establish a connection for the client application that is calling RemoteStopConnection
disconnectid	The ID of another client application that you want to disconnect

Return value

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

50 Distributed service error
52 Distributed communications error
55 Request terminated abnormally
56 Response to request incomplete
57 Not connected

Usage

To issue RemoteStopConnection, a client application must have administrative privileges. The server application grants connection privileges. The script for the server application's ConnectionBegin event can specify the privileges for each connection by returning a ConnectPrivilege value. A ConnectPrivilege value of ConnectWithAdminPrivilege! gives a client administrative privileges.

RemoteStopConnection is typically used in conjunction with the GetServerInfo function. A client application can use GetServerInfo to retrieve information about all client connections to a particular server application. Once the information has been retrieved, the client that issues the GetServerInfo function can then use RemoteStopConnection to disconnect one or more of the other clients.

Examples

In this example, a client application uses RemoteStopConnection to disconnect another client. The ID for the other client is obtained from a DataWindow control. The DataWindow control contains connection information that was retrieved by using the GetServerInfo function:

```
int li_rownum
string ls_id
```

```
long ll_rc

li_rownum = dw_connections.GetSelectedRow(0)

if li_rownum = 0 then
   MessageBox("Select Client","Select client first,
   then disconnect")
end if

ls_id = dw_connections.object.clientid[li_rownum]

ll_rc = MessageBox("Disconnect Client?","OK to
   disconnect client " + ls_id +
   "?",StopSign!,OKCancel!,2)

if ll_rc = 1 then
   myconnect.RemoteStopConnection(ls_id)
end if
```

See also

GetServerInfo

RemoteStopListening

Description

Allows a client application to instruct a server application to stop listening for client requests.

This function applies to distributed applications only.

Applies to

Connection objects

Syntax

connection.**RemoteStopListening** ( )

Argument	Description
connection	The name of the Connection object that identifies the server application you want to instruct to stop listening

Return value

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

50 Distributed service error
52 Distributed communications error
53 Requested server not active
55 Request terminated abnormally
56 Response to request incomplete
57 Not connected

Usage

To issue RemoteStopListening, a client application must have the ConnectWithAdminPrivilege! privilege.

Examples

In this example, a client application calls the RemoteStopListening function to instruct the server application identified by the myconnect Connection object to stop listening:

```
myconnect.RemoteStopListening()
```

See also

ConnectToServer

Repair

Description

Updates the target database with corrections that have been made in the pipeline user object's Error DataWindow.

Applies to

Pipeline objects

Syntax

pipelineobject.**Repair** (*destinationtrans*)

Argument	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

If any argument's value is NULL, Repair returns NULL.

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.

Examples

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*)

Argument	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. If any argument's value is NULL, Replace returns NULL.

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 the *DataWindow Reference*

ReplaceText

Description

Replaces selected text in an edit control with a specified string.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, and DropDownPictureListBox controls

Syntax

editname.**ReplaceText** (*string*)

Argument	Description
editname	The name of the DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, or DropDownPictureListBox control 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

Long. Returns the number of characters in *string* and -1 if an error occurs. If any argument's value is NULL, ReplaceText returns NULL.

Usage

If there is no selection, ReplaceText inserts the replacement text at the cursor position.

In a RichTextEdit control, the selection can include pictures.

Other ways to replace text

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 or DataStore. The values from the database are redisplayed in the row.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**ReselectRow** (*row*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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). If any argument's value is NULL, ReselectRow returns NULL.

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.

About timestamp support
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

Clears data from a control or object. The syntax you choose depends on the target object.

To	Use
Clear the data from a DataWindow control or DataStore	Syntax 1
Delete all items from a list	Syntax 2
Delete all the data (and optionally the series and categories) from a graph	Syntax 3
Return to the beginning of a trace file	Syntax 4

Syntax 1

For DataWindows and DataStores

Description

Clears all the data from a DataWindow control or DataStore object.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**Reset** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow you want to clear

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *dwcontrol* is NULL, Reset returns NULL. The return value is usually not used.

Usage

Reset 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.

If you call Reset when the Retrieve As Needed option is set, Reset will clear the rows that have been retrieved. However, because Retrieve As Needed is on, the DataWindow immediately retrieves the next set of rows. To prevent the rows from being retrieved, call DBCancel before calling Reset. If all the rows have been retrieved (the cursor has been closed and the RetrieveEnd event has occurred), then when Reset clears the DataWindow, it stays empty.

Examples

This statement completely clears the contents of dw_employee:

```
dw_employee.Reset()
```

In a DataWindow whose Retrieve As Needed option is on, this example cancels the retrieval before resetting the DataWindow:

```
dw_employee.DBCancel()
dw_employee.Reset()
```

See also

DeleteRow

Syntax 2

For listboxes

Description

Deletes all the items from a list.

Applies to

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

listboxname.**Reset ( )**

Argument	Description
listboxname	The name of the listbox control from which to delete all items

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *listboxname* is NULL, Reset returns NULL. The return value is usually not used.

Examples

This statement deletes all items in the ListBox portion of ddlb_Actions:

```
ddlb_Actions.Reset()
```

See also

DeleteItem

Syntax 3

For graphs

Description

Deletes the data, the categories, or the series from a graph.

Applies to

Graph controls in windows and user objects and graphs within a DataWindow object with an external data source.

Does not apply to other graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax

controlname.**Reset** (*graphresettype*)

Argument	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, Reset returns NULL. The return value is usually not used.

Usage

Use Reset to clear the data in a graph before you add new data.

Examples

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

Syntax 4 For trace files

Description

Goes back to the beginning of the trace file so you can begin rereading the file contents.

Applies to

TraceFile objects

Syntax

instancename.**Reset** ( )

Argument	Description
instancename	Instance name of the TraceFile object

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileNotOpenError!—The specified trace file has not been opened

Usage

Use this function to return to the start of the open trace file and begin rereading the contents of the file. To use the Reset function, you must have previously opened the trace file with the Open function. You use the Reset and Open functions as well as the other properties and functions provided by the TraceFile object to access the contents of a trace file directly. You use these functions if you want to perform your own analysis of the tracing data instead of using the available modeling objects.

Examples

This example returns you to the start of the open trace file ltf_file so that the file's contents can be reread:

```
TraceFile ltf_file
string ls_filename

ltf_file = CREATE TraceFile
ltf_file.Open(ls_filename)
...
ltf_file.Reset(ls_filename)
...
```

See also

Open
NextActivity
Close

ResetArgElements

Description

Clears the argument list.

Applies to

Window ActiveX controls

Syntax

activexcontrol.**ResetArgElements ( )**

Argument	Description
activexcontrol	Identifier for the instance of the PowerBuilder window ActiveX control. When used in HTML, this is the NAME attribute of the object element. When used in other environments, this references the control that contains the PowerBuilder window ActiveX

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function after calling InvokePBFunction or TriggerPBEvent to clear the argument list.

If you populate the argument list with SetArgElement, you should call this function to clear the argument list after using InvokePBFunction or TriggerPBEvent to call an event or function with arguments.

Examples

This JavaScript example calls the ResetArgElements function:

```
...
   retcd = PBRX1.TriggerPBEvent(theEvent, numargs);
   rc = parseInt(PBRX1.GetLastReturn());
   IF (rc != 1) {
      alert("Error. Empty string.");
   }
   PBRX1.ResetArgElements();
...
```

This VBScript example calls the ResetArgElements function:

```
...
   retcd = PBRX1.TriggerPBEvent(theEvent, numargs)
   rc = PBRX1.GetLastReturn()
   IF rc <> 1 THEN
      msgbox "Error. Empty string."
   END IF
   PBRX1.ResetArgElements()
...
```

See also

GetLastReturn
InvokePBFunction
SetArgElement
TriggerPBEvent

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 and DataStore objects

Syntax

controlname.**ResetDataColors** ({ *graphcontrol*, } *seriesnumber*, *datapointnumber*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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. If any argument's value is NULL, ResetDataColors returns NULL.

Default color for data points

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.FinSeries("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 or DataStore from using the programmer-specified transaction object that is currently in effect via a call to the SetTransObject function. After you call the ResetTransObject function, the DataWindow control or DataStore uses its internal transaction object.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**ResetTransObject** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow for 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. If *dwcontrol* is NULL, ResetTransObject returns NULL. 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 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.

Examples

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 a DataWindow or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**ResetUpdate** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If *dwcontrol* is NULL, ResetUpdate returns NULL.

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 or DataStore, 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.

Examples

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 properties and then redraws the object.

Applies to

Any object, except a child DataWindow

Syntax

objectname.**Resize** (*width, height*)

Argument	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 or if *objectname* is a minimized or maximized window. If any argument's value is NULL, Resize returns NULL.

Usage

You cannot use Resize for a child DataWindow.

Resize does not resize a minimized or maximized sheet or window. If the window is minimized or maximized, Resize returns –1.

Equivalent syntax You can set object's Width and Height properties instead of calling the Resize function. 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 properties of gb_box1 and redraws gb_box1 with the new properties:

```
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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

RespondRemote (*boolean*)

Argument	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). If *boolean* is NULL, RespondRemote returns NULL.

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 INFO For more information about PowerBuilder as a client, see OpenChannel and ExecRemote. For more information about PowerBuilder as a server, see StartServerDDE.

Examples

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.

Examples

In the application-level script for the Idle event, this statement restarts the application:

```
Restart ( )
```

See also

HALT on page 147

Retrieve

Description

Retrieves rows from the database for a DataWindow control or DataStore. 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

dwcontrol.**Retrieve** ({, *argument*, *argument* . . . })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow that you want to cause to retrieve rows from the database
argument (optional)	One or more values that you want to use as retrieval arguments in the SQL SELECT statement defined in *dwcontrol*

Return value

Long. Returns the number of rows displayed (that is, rows in the primary buffer) if it succeeds and -1 if it fails. If any argument's value is NULL, Retrieve returns NULL.

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 or DataStore, 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 or DataStore are discarded and replaced with the retrieved rows. You can return the code 2 in the RetrieveStart event to prevent this. In this case, Retrieve adds any retrieved rows to the ones that are already in the buffers. (See the last example.)

If arguments are expected but not specified, the user is prompted for the retrieval arguments.

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
```

Modifying the SQL SELECT statement
:Salary and :Region are declared as arguments in the DataWindow painter. To modify the SQL SELECT statement, choose DesignÆData Source in the DataWindow painter and then choose EditÆRetrieval Arguments in the SQL 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_1.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:

```
RETURN 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 return code in the RetrieveStart event:

```
dw_1.Reset( )
dw_1.Retrieve(100)
dw_1.Retrieve(200)
```

See also

DeleteRow
InsertRow
SetTrans
SetTransObject

Reverse

Description

Reverses the order or characters in a string.

Syntax

Reverse (*string*)

Argument	Description
string	A string whose characters you want to reorder so that the last character is first and the first character is last

Return value

String. Returns a string with the characters of *string* in reversed order. Returns the empty string if it fails.

Usage

Reverse is useful with the IsArabic and IsHebrew functions, which help you implement RightToLeft character display when you are using RightToLeft versions of PowerBuilder and Windows.

Examples

Under the RightToLeft version of Windows, this statement returns a string with the characters in reverse order from the characters entered in sle_name:

```
string ls_name
ls_name = Reverse(sle_name.Text)
```

See also

IsArabic
IsHebrew

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*)

Argument	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. If any argument's value is NULL, RGB returns NULL.

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.

About color values

The value of a component of a 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

Color	Red value	Green value	Blue value
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 properties 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)
```

See also

RGB in the *DataWindow Reference*

Right

Description

Obtains a specified number of characters from the end of a string.

Syntax

Right (*string, n*)

Argument	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 any argument's value is NULL, Right returns NULL.

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 the *DataWindow Reference*

RightTrim

Description

Removes spaces from the end of a string.

Syntax

RightTrim (*string*)

Argument	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. If any argument's value is NULL, RightTrim returns NULL.

Examples

This statement returns RUTH:

```
RightTrim("RUTH ")
```

See also

LeftTrim
Trim
RightTrim in the *DataWindow Reference*

Round

Description

Rounds a number to the specified number of decimal places.

Syntax

Round (*x*, *n*)

Argument	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 or if any argument's value is NULL.

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 the *DataWindow Reference*

RoutineList

Description

Provides a list of the routines included in a performance analysis model.

Applies to

ProfileClass and Profiling objects

Syntax

instancename.**RoutineList** (*list*)

Argument	Description
instancename	Instance name of the ProfileClass or Profiling object
list	An unbounded array variable of data type ProfileRoutine in which RoutineList stores a ProfileRoutine object for each routine that exists in the model within a class. This argument is passed by reference

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- ModelNotExistsError! —No model exists

Usage

Use this function to extract a list of the routines included in the performance analysis model in a particular class. You must have previously created the performance analysis model from a trace file using the BuildModel function. Each routine is defined as a ProfileRoutine object and provides the time spent in the routine, any called routines, the number of times each routine was called, and the class to which the routine belongs. The routines are listed in no particular order.

Object creation and destruction for a class are each indicated by a routine in this list as well as by embedded SQL statements.

Examples

This example lists the routines included in each class found in a performance analysis model:

```
Long ll_cnt
ProfileCall lproc_call[]

lpro_model.BuildModel()
lpro_model.RoutineList(iprort_list)
...
```

See also

ClassList

RowCount

Description

Obtains the number of rows that are currently available in a DataWindow control or DataStore. To determine the number of rows available, the RowCount function checks the primary buffer.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**RowCount** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol*, 0 if no rows are currently available, and -1 if an error occurs. If *dwcontrol* is NULL, RowCount returns NULL.

Usage

The primary buffer for a DataWindow control or DataStore contains the rows that are currently available for display or printing. 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.

Examples

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 property 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 (or DataStore object) to another, or from one buffer to another within a single DataWindow control (or DataStore).

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**RowsCopy** (*startrow, endrow, copybuffer, targetdw, beforerow, targetbuffer*)

Argument	Description
dwcontrol	The name of a DataWindow control, DataStore, 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
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 or DataStore object to which you want to copy the rows. *Targetdw* can be the same DataWindow (or DataStore) or another DataWindow (or DataStore)
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. If any argument's value is NULL, RowsCopy returns NULL.

Usage

When you use the RowsCopy function, the status of the rows that are copied to the primary buffer is NewModified!. If you issue an update request, PowerBuilder will send SQL INSERT statements to the DBMS for the new rows.

When you use RowsCopy, data is not automatically retrieved for drop-down DataWindows in the target DataWindow or DataStore, as it is when you call InsertRow. You must explicitly call Retrieve for child DataWindows in the target.

Uses for RowsCopy include:

- Making copies of one or more rows so that the users can create new rows based on existing data
- Printing a range of rows by copying selected rows to another DataWindow and printing the second DataWindow

Examples

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!)
```

This example copies all the rows starting with the current row in dw_1 to the beginning of the primary buffer in the dropdown DataWindow state_id:

```
datawindowchild dwc
dw_3.GetChild("state_id", dwc)
dw_1.RowsCopy(dw_1.GetRow(), &
   dw_1.RowCount(), Primary!, dwc, 1, Primary!)
```

This example copies all the rows starting with the current row in dw_1 to the beginning of the primary buffer in the nested report d_employee:

```
datawindowchild dwc
dw_composite.GetChild("d_employee", dwc)
dw_1.RowsCopy(dw_1.GetRow(), &
   dw_1.RowCount(), Primary!, dwc, 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

dwcontrol **RowsDiscard** (*startrow, endrow, buffer*)

Argument	Description
dwcontrol	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. If any argument's value is NULL, RowsDiscard returns NULL.

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.

Examples

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 one DataWindow control (or DataStore) and inserts them in another. Alternatively, RowsMove moves rows from one buffer to another within a single DataWindow control (or DataStore).

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**RowsMove** (*startrow*, *endrow*, *movebuffer*, *targetdw*, *beforerow*, *targetbuffer*)

Argument	Description
dwcontrol	The name of a DataWindow control, DataStore, 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 or DataStore to which you want to move the rows. *Targetdw* can be the same DataWindow control (or DataStore) or a different DataWindow control (or DataStore), but it cannot be a child DataWindow
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. If any argument's value is NULL, RowsMove returns NULL.

Usage

When you use RowsMove, the rows have the status NewModified! in the target DataWindow.

If you move rows between buffers in a single DataWindow control or DataStore, 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 (or DataStore) to another and back again, the rows' status is NewModified! because they came from a different DataWindow.

When you use RowsMove, data is not automatically retrieved for drop-down DataWindows in the target DataWindow, as it is when you call InsertRow. You must explicitly call Retrieve for child DataWindows in the target.

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

Examples

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* })

Argument	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 On Macintosh, Minimized! launches the application in the background ♦ 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. If any argument's value is NULL, Run returns NULL.

Usage

You can use Run for 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:\PB\FILE1 %1
```

On 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")
```

On UNIX, this statement runs a data entry application in the employee directory. The parameter new is passed to the program:

```
Run( &
"/export/home/employee/dataentr.exe new")
```

Save

Description

Saves an OLE object in an OLE control or an OLE storage object.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Syntax

oleobject.**Save** ( )

Argument	Description
oleobject	The name of an OLE control or an OLE storage variable

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

If *oleobject* is NULL, Save returns NULL.

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 control, PowerBuilder saves the object in the OLE 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 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 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

Saves the contents of a DataWindow, DataStore, graph, OLE control, or OLE storage in a file. The syntax you use depends on the type of object you want to save.

To	Use
Save the contents of a DataWindow or DataStore	Syntax 1
Save the data in a graph	Syntax 2
Save the OLE object in an OLE control to a storage file	Syntax 3
Save the OLE object in an OLE control to a storage object in memory	Syntax 4
Save an OLE storage and any controls that have opened that storage in a file	Syntax 5
Save an OLE storage object in another OLE storage object	Syntax 6

Syntax 1

For DataWindows and DataStores

Description

Saves the contents of a DataWindow or DataStore in the format you specify.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SaveAs** ({ *filename*, *saveastype*, *colheading* })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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* or specify an empty string (""), PowerBuilder prompts for the filename **Working with DataStore objects** If you are working with a DataStore, you must supply the *filename* argument

Argument	Description
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: ♦ Clipboard! — Save to the clipboard ♦ CSV! — Comma-separated values ♦ dBASE2! — dBASE-II format ♦ dBASE3! — dBASE-III format ♦ DIF! — Data Interchange Format ♦ Excel! — Microsoft Excel format ♦ Excel5! — Microsoft Excel 5 format ♦ HTMLTable! — Text with HTML formatting that approximates the DataWindow layout ♦ PSReport! — Powersoft Report (PSR) 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 ♦ WMF! — Windows Metafile format **Platform information** On Macintosh, PSReport! and WMF! are not supported. The Clipboard! format does not support saving graphs to the clipboard.
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. **For dBASE files** Column headings are always saved for dBASE files.

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SaveAs returns NULL.

Usage

If you don't specify any arguments for SaveAs, PowerBuilder displays the Save As dialog box.

A dropdown listbox lets the user specify the format of the saved data.

If the DataWindow is a composite report, Report format (PSReport! value of SaveAsType) is the only reasonable choice.

If the DataWindow object has the RichText presentation style, choosing PSReport! has no effect.

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)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
dw_History.SaveAs("HD:Inventory:Employee History", &
CSV!, FALSE)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
dw_History.SaveAs( &
"/export/home/inventory/employee.his", &
CSV!, FALSE)
```

See also

Print
Update

Syntax 2

For graph objects

Description

Saves the data in a graph in the format you specify.

Applies to

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

Syntax

controlname.**SaveAs** ({ *filename*, } { *graphcontrol*, *saveastype*, *colheading* })

Argument	Description
controlname	The name of the graph control whose contents you want to save or the name of the DataWindow DataStore containing the graph

Argument	Description
filename (optional)	A string whose value is the name of the file in which you want to save the data in the graph. If you omit *filename* or specify an empty string (""), PowerBuilder prompts the user for a filename
graphcontrol (DataWindow control only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore whose contents you want to save
saveastype (optional)	A value of the SaveAsType enumerated data type specifying the format in which to save the data represented in the graph. Values are: ♦ Clipboard! — Save an image of the graph to the clipboard ♦ CSV! — Comma-separated values ♦ dBASE2! — dBASE-II format ♦ dBASE3! — dBASE-III format ♦ DIF! — Data Interchange Format ♦ Excel! — Microsoft Excel format ♦ PSReport! — Powersoft Report (PSR) 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 ♦ WMF! — Windows Metafile format **Platform information** On Macintosh, Clipboard!, PSReport!, and WMF! are not supported.
colheading (optional)	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SaveAs returns NULL.

Usage

If you don't specify any arguments for SaveAs, PowerBuilder displays the Save As dialog box, letting the user specify the format of the saved data.

Examples

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)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
gr_History.SaveAs("HD:Human Resrc:Employee History", &
CSV!, FALSE)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
gr_History.SaveAs( &
"/export/home/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)
```

See also

Print

Syntax 3

For saving an OLE control to a file

Description

Saves the object in an OLE control in a storage file.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE controls

Syntax

olecontrol.**SaveAs** (*OLEtargetfile*)

Argument	Description
olecontrol	The name of the OLE 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

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

If any argument's value is NULL, SaveAs returns NULL.

Usage

The Open function establishes a connection between a storage file and a storage object, or a storage file or object and an OLE control. The Save function uses this connection to save the OLE data.

When you call SaveAs for an OLE 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.

Examples

This example saves the object in the control ole_1:

```
integer result
result = ole_1.SaveAs("c:\ole\expense.ole")
```

See also

Open
Save

Syntax 4 For saving an OLE control to an OLE storage

Description

Saves the object in an OLE control to an OLE storage object in memory.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE controls

Syntax

olecontrol.**SaveAs** (*targetstorage*, *substoragename*)

Argument	Description
olecontrol	The name of the OLE control containing the object you want to save
targetstorage	The name of an object variable of OLEStorage in which to store the object in *olecontrol*
substoragename	A string whose value is the name of a substorage within *targetstorage*. If *substorage* does not exist, SaveAs will create it

Return value

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

If any argument's value is NULL, SaveAs returns NULL.

Usage

The Open function establishes a connection between a storage file and a storage object, or a storage file or object and an OLE control. The Save function uses this connection to save the OLE data.

When you call SaveAs for an OLE 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.

Examples

This example saves the object in the control ole_1 in the storage variable stg_stuff:

```
integer result
result = ole_1.SaveAs(stg_stuff)
```

See also

Open
Save

Syntax 5 For saving an OLE storage object to a file

Description

Saves an OLE storage object to a file. If OLE controls have opened the OLE storage object, this syntax of SaveAs puts them in a saved state too.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE storage objects

Syntax

olestorage.**SaveAs** (*OLEtargetfile*)

Argument	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

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

If any argument's value is NULL, SaveAs returns NULL.

Usage

The Open function establishes a connection between a storage file and a storage object, or a storage file or object and an OLE control. The Save function uses this connection to save the OLE data.

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 INFO For information about the structure of storage files, see the Open function.

Examples

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:\ole\mystuff.ole")
```

This example opens a substorage in one file and saves it in another file. An OLE 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")
```

See also

Close
Open
Save

Syntax 6

For saving an OLE storage object in another OLE storage

Description

Saves an OLE storage object to another OLE storage object variable in memory.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE storage objects

Syntax

olestorage.**SaveAs** (*substoragename*, *targetstorage*)

Argument	Description
olestorage	The name of an object variable of type OLEStorage containing the OLE object you want to save

Argument	Description
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

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

If any argument's value is NULL, SaveAs returns NULL.

Usage

The Open function establishes a connection between a storage file and a storage object, or a storage file or object and an OLE control. The Save function uses this connection to save the OLE data.

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 INFO For information about the structure of storage files, see the Open function.

Examples

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
Save

SaveAsAscii

Description

Saves the contents of a DataWindow or DataStore into a standard ASCII text file.

Applies to

DataWindow controls and DataStore objects

Syntax

dwcontrol.**SaveAsAscii** (*filename* {, *separatorcharacter* {, *quotecharacter* {, *lineending* } } })

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore whose contents you want to save
filename	A string whose value is the name of the file in which to save the contents
separatorcharacter (optional)	A string whose value is the character to be used to delimit values. If you omit *separatorcharacter*, the default is a tab character
quotecharacter (optional)	A string whose value is the character to be used to wrap values. If you omit *quotecharacter*, the default is no character
lineending (optional)	A string whose value is placed at the end of each line. If you omit *lineending*, the default is a carriage return plus a newline character (~r~n)

Return value

Long. Returns 1 if it succeeds and -1 if an error occurs.

Usage

SaveAsAscii is a cross between the SaveAs (Text!) function and the SaveAs (HTMLTable!) function with additional arguments. It mirrors more closely what the user sees on the screen. Arguments allow the user to control how contents are separated and delimited in the ASCII file. PowerBuilder assigns a cell for each DataWindow object (which can include computed columns and group totals). If a cell is empty, PowerBuilder puts the *quotecharacter* between the *separatorcharacter* in the output file.

Examples

This statement saves the contents of dw_Quarter to the file H:\Q2\RESULTS.TXT. The saved file is ASCII with the ampersand (&) as the separator character, single quote (') as the character used to wrap values and the default line ending (~r~n). Computed columns are included with the saved information:

```
dw_Quarter.SaveAsAscii("H:\Q2\RESULTS.TXT","&","'")
```

See also

SaveAs

SaveDocument

Description

Saves the contents of a RichTextEdit control in a file. You can specify either rich-text format (RTF) or ASCII text format for the file.

Applies to

RichTextEdit controls

Syntax

rtename.**SaveDocument** (*filename* {, *filetype* })

Argument	Description
rtename	The name of the RichTextEdit control whose contents you want to save
filename	A string whose value is the name of the file to be saved. If the file already exists, the FileExists event is triggered
filetype (optional)	A value of the FileType enumerated data type specifying the format of the saved file. Values are: ♦ FileTypeRichText! — Save the file in rich text format (RTF) ♦ FileTypeText! — Save the file as ASCII text

Return value

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

Usage

SaveDocument triggers a FileExists event when the file you name already exists.

Examples

This code for a CommandButton saves the document in the RichTextEdit rte_1:

```
integer li_rtn

li_rtn = rte_1.SaveDocument("c:\test.rtf", &
FileTypeRichText!)
```

If the file TEST.RTF already exists, PowerBuilder triggers the FileExists event with the following script. OpenWithParm displays a response window that asks the user if it is OK to overwrite the file. The return value from FileExists determines whether the file is saved:

```
OpenWithParm( w_question, &
"The specified file already exists. " + &
"Do you want to overwrite it?" )
IF Message.StringParm = "Yes" THEN
RETURN 0  // File is saved
ELSE
RETURN -1 // Saving is canceled
END IF
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
li_rtn = rte_1.SaveDocument("HD:Rich Text Test", &
FileTypeRichText!)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
li_rtn = rte_1.SaveDocument("/export/home/test.rtf", &
FileTypeRichText!)
```

See also

InsertDocument

Scroll

Description

Scrolls a multiline edit control or the edit control of a DataWindow a specified number of lines up or down.

Applies to

DataWindow, MultiLineEdit, and RichTextEdit controls

Syntax

editname.**Scroll** (*number*)

Argument	Description
editname	The name of the DataWindow, RichTextEdit, or MultiLineEdit in which you want to scroll up or down. If *editname* is a DataWindow, then Scroll affects its edit control
number	A long specifying the direction and number of lines you want to scroll. To scroll down, use a positive long value. To scroll up, use a negative long value

Return value

Long. For RichTextEdit controls, Scroll returns 1 if it succeeds. For other controls, Scroll returns the line number of the first visible line in *editname* if it succeeds. Scroll returns -1 if an error occurs. If any argument's value is NULL, Scroll returns NULL.

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 or a RichTextEdit:

ScrollNextPage
ScrollNextRow
ScrollPriorPage
ScrollPriorRow
ScrollToRow

ScrollNextPage

Scrolls to the next page in a DataWindow or RichTextEdit control.

To scroll	Use
To the next group of rows in a DataWindow (when the DataWindow does not have the RichTextEdit presentation style)	Syntax 1
A RichTextEdit control or RichTextEdit DataWindow to view the next page within the document	Syntax 2

Syntax 1 For DataWindow controls and child DataWindows

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

dwcontrol.**ScrollNextPage** ( )

Argument	Description
dwcontrol	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. If *dwcontrol* is NULL, ScrollNextPage returns NULL.

Usage

ScrollNextPage does not highlight the current row. Use SelectRow to let the user know what row is current.

FOR INFO 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

Examples

This statement scrolls dw_employee forward one page:

```
dw_employee.ScrollNextPage()
```

See also

Scroll
ScrollNextRow
ScrollPriorPage
ScrollPriorRow
ScrollToRow
SelectRow

Syntax 2 For RichTextEdit controls and DataWindows

Description

Scrolls to the next page of the document in a RichTextEdit control or RichTextEdit DataWindow.

Applies to

DataWindow and RichTextEdit controls

Syntax

rtename.**ScrollNextPage** ( )

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control in which you want to scroll to the next page. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow

Return value

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

Usage

When the RichTextEdit control shares data with a DataWindow, the RichTextEdit contains multiple instances of the document, one instance for each row.

When the last page of the document for one row is visible, calling ScrollNextPage advances to the first page for the next row.

Examples

This statement scrolls to the next page of the document in the RichTextEdit control rte_1. If there are multiple instances of the document, it can scroll to the next instance:

```
rte_1.ScrollNextPage()
```

See also

Scroll
ScrollNextRow
ScrollPriorPage
ScrollPriorRow

ScrollNextRow

Scrolls to the next row in a DataWindow or RichTextEdit control.

To scroll	Use
To the next row in a DataWindow, making the row current (when the DataWindow does not have the RichTextEdit presentation style)	Syntax 1
To the next instance of a document associated with a row in a RichTextEdit control or RichTextEdit DataWindow	Syntax 2

Syntax 1 For DataWindow controls and child DataWindows

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

dwcontrol.**ScrollNextRow** ( )

Argument	Description
dwcontrol	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. If *dwcontrol* is NULL, ScrollNextRow returns NULL.

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

Examples

This statement scrolls dw_employee to the next row:

```
dw_employee.ScrollNextRow()
```

See also

Scroll
ScrollNextPage
ScrollPriorPage
ScrollPriorRow
ScrollToRow
SelectRow

Syntax 2 For RichTextEdit controls and DataWindows

Description

Scrolls to the next instance of the document in a RichTextEdit control or RichTextEdit DataWindow. A RichTextEdit control has multiple instances of its document when it shares data with a DataWindow. The next instance of the document is associated with the next row in the DataWindow.

Applies to

DataWindow and RichTextEdit controls

Syntax

rtename.**ScrollNextRow** ( )

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control in which you want to scroll to the next document instance. Each instance is associated with a DataWindow row. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow

Return value

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

Usage

When the RichTextEdit shares data with a DataWindow, the RichTextEdit contains multiple instances of the document, one instance for each row.

ScrollNextRow advances to the next instance of the RichTextEdit document. In contrast, repeated calls to ScrollNextPage advance through all the pages of the document instance and then on to the pages for the next row.

Examples

This statement scrolls to the next instance of the document in the RichTextEdit control rte_1. (Each document instance is associated with a row of data):

```
rte_1.ScrollNextRow()
```

See also

Scroll
ScrollNextPage
ScrollPriorPage
ScrollPriorRow

ScrollPriorPage

Scrolls to the prior page in a DataWindow or RichTextEdit control.

To scroll	Use
To the prior group of rows in a DataWindow (when the DataWindow does not have the RichTextEdit presentation style)	Syntax 1
A RichTextEdit control or RichTextEdit DataWindow to view the prior page within the document	Syntax 2

Syntax 1 For DataWindow controls and child DataWindows

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

dwcontrol.**ScrollPriorPage** ( )

Argument	Description
dwcontrol	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. If *dwcontrol* is NULL, ScrollPriorPage returns NULL.

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

Examples

This statement scrolls dw_employee backward one page:

```
dw_employee.ScrollPriorPage()
```

See also

Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorRow
ScrollToRow
SelectRow

Syntax 2 For RichTextEdit controls and DataWindows

Description

Scrolls to the prior page of the document in a RichTextEdit control or RichTextEdit DataWindow.

Applies to

DataWindow and RichTextEdit controls

Syntax

rtename.**ScrollPriorPage** ( )

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control in which you want to scroll to the prior page. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow

Return value

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

Usage

When the RichTextEdit shares data with a DataWindow, the RichTextEdit contains multiple instances of the document, one instance for each row.

When the first page of the document for one row is visible, calling ScrollPriorPage goes to the last page for the prior row.

Examples

This statement scrolls to the prior page of the document in the RichTextEdit control rte_1. If there are multiple instances of the document, it can scroll to the prior instance:

```
rte_1.ScrollPriorPage()
```

See also

Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorRow

ScrollPriorRow

Scrolls to the prior row in a DataWindow or RichTextEdit control.

To scroll	Use
To the prior row in a DataWindow, making the row current (when the DataWindow does not have the RichTextEdit presentation style)	Syntax 1
To the prior instance of a document associated with a row in a RichTextEdit control or RichTextEdit DataWindow	Syntax 2

Syntax 1 For DataWindow controls and child DataWindows

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

dwcontrol.**ScrollPriorRow** ( )

Argument	Description
dwcontrol	The name of the DataWindow 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. If *dwcontrol* is NULL, ScrollPriorRow returns NULL.

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

Examples

This statement scrolls dw_employee to the prior row:

```
dw_employee.ScrollPriorRow()
```

See also

Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorPage
ScrollToRow
SelectRow

Syntax 2 For RichTextEdit controls and DataWindows

Description

Scrolls to the prior instance of the document in a RichTextEdit control or RichTextEdit DataWindow. A RichTextEdit control has multiple instances of its document when it shares data with a DataWindow. The next instance of the document is associated with the next row in the DataWindow.

Applies to

DataWindow and RichTextEdit controls

Syntax

rtename.**ScrollPriorRow** ( )

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control in which you want to scroll to the prior document instance. Each instance is associated with a DataWindow row. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow

Return value

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

Usage

When the RichTextEdit shares data with a DataWindow, the RichTextEdit contains multiple instances of the document, one instance for each row.

ScrollPriorRow goes to the prior instance of the RichTextEdit document. In contrast, repeated calls to ScrollPriorPage pages back through all the pages of the document instance and then back to the pages for the prior row.

Examples

This statement scrolls to the prior instance of the document in the RichTextEdit control rte_1. (Each document instance is associated with a row of data):

```
rte_1.ScrollPriorRow()
```

See also

Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorPage

ScrollToRow

Scrolls to a specified row or document instance.

To scroll	Use
A DataWindow to the specified row	Syntax 1
A RichTextEdit to the document instance associated with the specified row	Syntax 2

Syntax 1

For DataWindow controls and child DataWindows

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

dwcontrol.**ScrollToRow** (*row*)

Argument	Description
dwcontrol	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. If any argument's value is NULL, ScrollToRow returns NULL.

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

Examples

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

Syntax 2 For RichTextEdit controls

Description

Scrolls to the document instance associated with the specified row when the RichTextEdit controls shares data with a DataWindow.

Applies to

RichTextEdit controls

Syntax

rtename.**ScrollToRow** (*row*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to scroll to a document instance associated with the specified 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 number of rows in the associated DataWindow, it scrolls to the last row

Return value

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

Usage

When the RichTextEdit shares data with a DataWindow, the RichTextEdit contains multiple instances of the document, one instance for each row. ScrollToRow goes to the instance associated with the specified row.

Examples

In this example, dw_1 has retrieved at least 25 rows of data. After calling DataSource, the RichTextEdit control contains at least 25 instances of its document. ScrollToRow scrolls to the 25th instance:

```
rte_1.DataSource(dw_1)
rte_1.ScrollToRow(25)
```

See also

Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorPage
ScrollPriorRow

Second

Description

Obtains the number of seconds in the seconds portion of a time value.

Syntax

Second (*time*)

Argument	Description
time	The time value from which you want the seconds

Return value

Integer. Returns the seconds portion of *time* (00 to 59). If *time* is NULL, Second returns NULL.

Examples

This statement returns 31:

```
Second(19:01:31)
```

See also

Hour
Minute
Second in the *DataWindow Reference*

SecondsAfter

Description

Determines the number of seconds one time occurs after another.

Syntax

SecondsAfter (*time1*, *time2*)

Argument	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. If any argument's value is NULL, SecondsAfter returns NULL.

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
RelativeDate
RelativeTime
SecondsAfter in the *DataWindow Reference*

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.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLEStream objects

Syntax

olestream.**Seek** (*position* {, *origin* })

Argument	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

If any argument's value is NULL, Seek returns NULL.

Examples

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:\ole\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

SelectedColumn

Description

Obtains the number of the character column just after the insertion point in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**SelectedColumn** ( )

Argument	Description
rtename	The name of the RichTextEdit in which you want the number of the character after the insertion point

Return value

Integer. Returns the number of the character before the insertion point in *rtename*. If an error occurs, SelectedColumn returns -1.

Usage

The insertion point can be at the beginning or end of the selection. Therefore, SelectedColumn can return the first character of the selection or the character just after the selection, depending on the position of the insertion point.

Examples

If the insertion point is positioned before the fifth character on line 8 of the RichTextEdit rte_Contact, the following example sets li_SL to 5 and li_line to 8:

```
integer li_SL, li_line
li_SL = rte_Contact.SelectedColumn()
```

See also

LineLength
Position
SelectedLine
SelectedPage
SelectedText
TextLine

SelectedIndex

Description

Obtains the number of the selected item in a ListBox or ListView control.

Applies to

ListBox and ListView controls

Syntax

listcontrolname.**SelectedIndex ( )**

Argument	Description
listcontrolname	The name of the ListBox or ListView control in which you want to locate the selected item

Return value

Integer. Returns the index of the selected item in *listcontrolname*. 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. If *listcontrolname* is NULL, SelectedIndex returns NULL.

Usage

SelectedIndex and SelectedItem are meant for lists that allow a single selection only (when the MultiSelect property for the control is FALSE).

When the MultiSelect property 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, then this example sets li_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 and PictureListBox controls

Syntax

listboxname.**SelectedItem** ( )

Argument	Description
listboxname	The name of the ListBox or PictureListBox 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. If *listboxname* is NULL, SelectedItem returns NULL.

Usage

SelectedIndex and SelectedItem are meant for lists that allow a single selection only (when the MultiSelect property for the control is FALSE).

When the MultiSelect property 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.

Examples

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 line endings.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, and DropDownPictureListBox controls

Syntax

editname.**SelectedLength ( )**

Argument	Description
editname	The name of the DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, or DropDownPictureListBox control 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

Long. Returns the length of the selected text in *editname*. If no text is selected, SelectedLength returns 0. If an error occurs, it returns -1. If *editname* is NULL, SelectedLength returns NULL.

Usage

The characters that make up a line ending, produced by typing CTRL+ENTER or ENTER, is different on different platforms. On Windows, it is a carriage return plus a line feed and equals two characters when calculating the length. On other platforms, a line ending is a single character. A line that has wrapped has no line-ending character.

For DropDownListBox and DropDownPictureListBox controls, SelectedLength returns -1 if the control's AllowEdit property is set to FALSE.

Focus and the selection in a dropdown listbox
When a DropDownListBox or DropDownPictureListBox 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
SelectedPage
SelectedStart
TextLine

SelectedLine

Description

Obtains the number of the line that contains the insertion point in an editable control.

Applies to

DataWindow, MultiLineEdit, and RichTextEdit controls

Syntax

editname.**SelectedLine** ( )

Argument	Description
editname	The name of the DataWindow, MultiLineEdit, or RichTextEdit 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

Long. Returns the number of the line containing the insertion point in *editname*. If an error occurs, SelectedLine returns -1. If *editname* is NULL, SelectedLine returns NULL.

Usage

For EditMask controls, SelectedLine will compile but always returns 1.

The insertion point can be at the beginning or end of the selection. Therefore, SelectedLine can return the first or last selected line, depending on the position of the insertion point.

Examples

If the insertion point 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
SelectedColumn
SelectedPage
SelectedText
TextLine

SelectedPage

Description

Obtains the number of the current page in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**SelectedPage** ( )

Argument	Description
rtename	The name of the RichTextEdit control in which you want the number of the current page

Return value

Integer. Returns the number of the current page in *rtename*. If an error occurs, SelectedPage returns -1.

Usage

The current page in a RichTextEdit control is the page that contains the insertion point in text entry mode or the page currently being displayed in preview mode.

When the RichTextEdit shares data with a DataWindow, SelectedPage returns the page number within the document instance for the current row.

FOR INFO For more information about document instances, see DataSource.

Examples

This example returns the page number of the current page:

```
integer li_pagect
li_pagect = rte_1.SelectedPage()
```

See also

DataSource
PageCount
Preview
SelectedLength
SelectedLine
SelectedStart
SelectedText

SelectedStart

Description

Reports the position of the first selected character in an editable control.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, and DropDownPictureListBox controls

Syntax

editname.**SelectedStart** ( )

Argument	Description
editname	The name of the DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, or DropDownPictureListBox control 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

Long. 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 insertion point. If an error occurs, SelectedStart returns -1. If *editname* is NULL, SelectedStart returns NULL.

Usage

For all controls except RichTextEdit, SelectedStart counts from the start of the text and includes spaces and line endings.

For RichTextEdit controls, SelectedStart counts from the start of the line on which the selection begins. The start is at the opposite end of the selection from the insertion point. For example, if the user dragged back to make the selection, the start of the selection is at the end of the highlighted text and the insertion point is before the start. Use the Position function to get information about the start *and* end of the selection.

Focus and the selection in a dropdown listbox
When a DropDownListBox or DropDownPictureListBox loses focus, the selected text is no longer selected.

Examples

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
SelectedPage

SelectedText

Description

Obtains the selected text in an editable control.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, and DropDownPictureListBox controls

Syntax

editname.**SelectedText** ( )

Argument	Description
editname	The name of the DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, or DropDownPictureListBox control from which you want the selected text. For a DropDownListBox or DropDownPictureListBox, the AllowEdit property 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 (""). If *editname* is NULL, SelectedText returns NULL.

Usage

In a RichTextEdit control, any pictures in the selection are ignored. If the selection contains input fields, the names of the input fields, enclosed in brackets, become part of the string SelectedText returns. The contents of the input fields are not returned.

For example, when the salutation of a letter is selected, SelectedText might return:

```
Dear {title} {lastname}:
```

Focus and the selection in a dropdown listbox
When a DropDownListBox or DropDownPictureListBox 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
```

See also SelectText

SelectItem

Finds and highlights an item in a ListBox, DropDownListBox, or TreeView control.

To select an item	Use
In a listbox control when you know the text of the item, but not its position	Syntax 1
In a listbox control when you know the position of the item in the control's list, or to clear the current selection	Syntax 2
In a TreeView control	Syntax 3

Syntax 1 When you know the text of an item

Description

Finds and highlights an item in a listbox when you can specify some or all of the text of the item.

Applies to

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

listboxname.**SelectItem** (*item, index*)

Argument	Description
listboxname	The name of the listbox control 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

Integer. Returns the index number of the selected item. If no match is found, SelectItem returns 0; it returns -1 if an error occurs. If any argument's value is NULL, SelectItem returns NULL.

Usage

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 or PictureListBox whose MultiSelect property 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 dropdown listbox To clear the edit box of a DropDownListBox or DropDownPictureListBox that the user cannot edit, use Syntax 2 of SelectItem.

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)
```

See also

AddItem
DeleteItem
FindItem
InsertItem
SetState

Syntax 2

When you know the item number

Description

Finds and highlights an item in a listbox when you can specify the index number of the item. You can also clear the selection by specifying zero as the index number.

Applies to

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

listboxname.**SelectItem** (*itemnumber*)

Argument	Description
listboxname	The name of the listbox control in which you want to select an item

Argument	Description
itemnumber	An integer whose value is the location (index) of the item in the listbox or the listbox portion of the dropdown listbox. Specify 0 for *itemnumber* to clear the selected item. For a ListBox or PictureListBox, 0 removes highlighting from the selected item. For a DropDownListBox or DropDownPictureListBox, 0 clears the textbox

Return value

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 clear the selected item. It returns -1 if an error occurs. If any argument's value is NULL, SelectItem returns NULL.

Usage

To find an item but not select it, use the FindItem function.

MultiSelect listboxes SelectItem has no effect on a ListBox or PictureListBox whose MultiSelect property is TRUE. Instead, use SetState to select items without affecting the selected state of other items in the list.

Clearing the textbox of a dropdown listbox To clear the textbox of a DropDownListBox or DropDownPictureListBox that the user cannot edit, set *itemnumber* to 0. Setting the control's text to the empty string doesn't work if the control's AllowEdit property is FALSE.

Examples

This example highlights item number 5:

```
integer li_Index
li_Index = lb_Actions.SelectItem(5)
```

This example clears the selection from the textbox 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

Syntax 3 For TreeView controls

Description

Selects a specified item.

Applies to

TreeView controls

Syntax

treeviewname.**SelectItem** (*itemhandle*)

Argument	Description
treeviewname	The name of the TreeView control in which you want to select an item
itemhandle	The handle of the specified item

Return value

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

Usage

Use the FindItem function to get handles for items at specific positions in the TreeView control.

Examples

This example selects the parent of the current TreeView item:

```
long ll_tvi, ll_tvparent
int li_tvret

ll_tvi = tv_list.FindItem(CurrentTreeItem! , 0)
ll_tvparent = tv_list.FindItem(ParentTreeItem! &
ll_tvi)

li_tvret = tv_list.SelectItem(ll_tvparent)
```

See also

FindItem

SelectObject

Description

Selects or clears the object in an OLE control but does not activate the server application. The server's menus are added to the PowerBuilder application's menus.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLE controls

Syntax

olecontrol.**SelectObject** (*selectstate*)

Argument	Description
olecontrol	The name of the OLE 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

If any argument's value is NULL, SelectObject returns NULL.

Examples

This example selects the object in the OLE control ole_1:

```
integer result
result = ole_1.SelectObject(TRUE)
```

SelectRow

Description

Highlights or unhighlights rows in a DataWindow control or DataStore. 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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SelectRow** (*row, boolean*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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) so that they are highlighted ♦ FALSE — Deselect the row(s) so that they are not highlighted

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SelectRow returns NULL.

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(row, TRUE)
```

SelectTab

Description

Selects the specified tab, displaying its tab page in the Tab control.

Applies to

Tab controls

Syntax

tabcontrolname.**SelectTab** (*tabidentifier*)

Argument	Description
tabcontrolname	The name of the Tab control in which you want to select a tab
tabidentifier	The tab you want to select. You can specify: ♦ The tab page index (an integer) ♦ The name of the user object (data type DragObject or UserObject) ♦ A string holding the name of the user object

Return value

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

Usage

Equivalent syntax You can select a tab by setting the SelectedTab property to the tab's index number:

```
tab_1.SelectedTab = 3
```

Examples

These three examples will select the third tab in tab_1. They could be in the script for a CommandButton on the window containing the Tab control tab_1:

```
tab_1.SelectTab(3)
```

```
tab_1.SelectTab(tab_1.uo_3)
```

```
string ls_tabpage
ls_tabpage = "uo_3"
tab_1.SelectTab(ls_tabpage)
```

This example opens an instance of the user object uo_fontsettings as a tab page and selects it:

```
userobject uo_tabpage
string ls_tabpage
ls_tabpage = "uo_fontsettings"
tab_1.OpenTab(uo_tabpage, ls_tabpage, 0)
tab_1.SelectTab(uo_tabpage)
```

See also

OpenTab

SelectText

Selects text in an editable control.

To select text in	Use
Any editable control, other than a RichTextEdit	Syntax 1
A RichTextEdit control or a DataWindow whose object has the RichTextEdit presentation style	Syntax 2

Syntax 1 For editable controls (except RichTextEdit)

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, DropDownListBox, and DropDownPictureListBox controls

Syntax

editname.**SelectText** (*start*, *length*)

Argument	Description
editname	The name of the DataWindow, EditMask, MultiLineEdit, SingleLineEdit, DropDownListBox, or DropDownPictureListBox control in which you want to select text
start	A long specifying the position at which you want to start the selection
length	A long specifying 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

Long. Returns the number of characters selected. If an error occurs, SelectText returns -1. If any argument's value is NULL, SelectText returns NULL.

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.

How much to select
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.

To select text in a DataWindow with the RichTextEdit presentation style, use Syntax 2.

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, returns 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

Syntax 2

For RichTextEdit controls and presentation styles

Description

Selects text beginning and ending at a line and character position in a RichTextEdit control.

Applies to

RichTextEdit and DataWindow controls

Syntax

rtename.**SelectText** (*fromline*, *fromchar*, *toline*, *tochar* { *band* })

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control in which you want to select text. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow
fromline	A long specifying the line number where the selection starts
fromchar	A long specifying the number in the line of the first character in the selection
toline	A long specifying the line number where the selection ends. To specify an insertion point, set *toline* and *tochar* to 0
tochar	A long specifying the number in the line of the character before which the selection ends
band (optional)	A value of the Band enumerated data type specifying the band in which to make the selection. Values are: ♦ Detail! ♦ Header! ♦ Footer! The default is the band that contains the insertion point

Return value

Long. Returns the number of characters selected. If an error occurs it returns -1. If any argument's value is NULL, SelectText returns NULL.

Usage

The insertion point is at the "to" end of the selection, that is, the position specified by *toline* and *tochar*. If *toline* and *tochar* are before *fromline* and *fromchar*, then the insertion point is at the beginning of the selection.

You cannot specify 0 for a character position when making a selection.

You cannot always use the values returned by Position to make a selection. Position can return a character position of 0 when the insertion point is at the beginning of a line.

To select an entire line, set the insertion point and call SelectTextLine. To select the rest of a line, set the insertion point and call SelectText with a character position greater than the line length.

Examples

This statement selects text from the first character in the RichTextEdit control to the fourth character on the third line:

```
rte_1.SelectText(1,1, 3,4)
```

This statement sets the insertion point at the beginning of line 2:

```
rte_1.SelectText(2,1, 0,0)
```

This example sets the insertion point at the end of line 2 by specifying a large number of characters. The selection highlight will extend past the end of the line:

```
rte_1.SelectText(2,999, 0,0)
```

This example sets the insertion point at the end of line 2 by finding out how long the line really is. The code moves the insertion point to the beginning of the line, gets the length, and then sets the insertion point at the end:

```
long ll_length

//Make line 2 the current line
rte_1.SelectText(2,1, 0,0)

// Specify a position after the last character
ll_length = rte_1.LineLength() + 1

// Set the insertion point at the end
rte_1.SelectText(2,ll_length, 0,0)
rte_1.SetFocus()
```

This example selects the text from the insertion point to the end of the current line. If the current line is the last line, the reported line length is 1 greater than the number of character you can select, so the code adjusts for it:

```
long ll_insertline, ll_insertchar
long ll_line, ll_count

// Get the insertion point
rte_1.Position(ll_insertline, ll_insertchar)

// Get the line number and line length
ll_line = rte_1.SelectedLine()
ll_count = rte_1.LineLength()
// Line length includes the eof file character,
// which can't be selected
IF ll_line = rte_1.LineCount() THEN ll_count -= 1
```

```
// Select from the insertion point to the end of
// line
rte_1.SelectText(ll_insertline, ll_insertchar, &
ll_line, ll_count)
```

See also

SelectedText
SelectTextAll
SelectTextLine
SelectTextWord

SelectTextAll

Description

Selects all the contents of a RichTextEdit control.

Applies to

RichTextEdit and DataWindow controls

Syntax

rtename.**SelectTextAll** ({ *band* })

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control in which you want to select all the contents. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow
band (optional)	A value of the Band enumerated data type specifying the band in which you want to select all the text. Values are: ♦ Detail! ♦ Header! ♦ Footer! The default is the band that contains the insertion point

Return value

Integer. Returns the number of characters selected. If an error occurs, SelectTextAll returns -1.

Examples

This statement selects all the text in the detail band:

```
rte_1.SelectTextAll()
```

This statement selects all the text in the header band:

```
rte_1.SelectTextAll(Header!)
```

See also

SelectedText
SelectText
SelectTextLine
SelectTextWord

SelectTextLine

Description

Selects the line containing the insertion point in a RichTextEdit control.

Applies to

RichTextEdit and DataWindow controls

Syntax

rtename.**SelectTextLine** ( )

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control in which you want select a line. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow

Return value

Integer. Returns the number of characters selected if it succeeds and -1 if an error occurs.

Usage

If the RichTextEdit control contains a selection, the insertion point is either at the beginning or end of the selection. The way the text was selected determines which. If the user made the selection by dragging toward the end, then calling SelectTextLine selects the line at the end of the selection. If the user dragged back, then SelectTextLine selects the line at the beginning of the selection.

SelectTextLine does not select the line-ending characters (carriage return and linefeed in Windows, carriage return on Macintosh, or linefeed on UNIX).

Examples

This statement selects the current line:

```
rte_1.SelectTextLine()
```

See also

SelectedText
SelectText
SelectTextAll
SelectTextWord

SelectTextWord

Description

Selects the word containing the insertion point in a RichTextEdit control.

Applies to

RichTextEdit and DataWindow controls

Syntax

rtename.**SelectTextWord** ( )

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control in which you want to select a word. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow

Return value

Integer. Returns the number of characters selected if it succeeds and -1 if a word cannot be selected or an error occurs.

Usage

A word is any group of alphanumeric characters and the following space, if any. If a space follows the selected word, the space is selected too. A word doesn't include punctuation and special characters such as $ or #. If punctuation or special characters follow the selected word, they are not selected.

If the character after the insertion point is a space, punctuation, special character, or end-of-line mark, SelectTextWord does not select anything and returns -1.

Examples

The following statement selects the word containing the insertion point:

```
rte_1.SelectTextWord()
```

This example selects the word at the insertion point. If there is no word, it increments the position until it finds a word. It checks when it reaches the end of a line and wraps to the next line as it looks for a word. If this script is assigned to a command button and the button is clicked repeatedly, you step through the text word by word:

```
integer li_rtn
long llstart, lcstart, ll_lines, ll_chars

ll_lines = rte_1.LineCount()
ll_chars = rte_1.LineLength()

li_rtn = rte_1.SelectTextWord()

// -1 if a word is not found at the insertion point
   DO WHILE li_rtn = -1
```

```
        // Get the position of the cursor
        rte_1.Position(llstart, lcstart)

        // Increment by 1 to look for next word
        lcstart += 1
        // If at end of line move to next line
        IF lcstart >= ll_chars THEN
           lcstart = 1  // First character
           llstart += 1 // next line

           // If beyond last line, return
           IF llstart > ll_lines THEN
              RETURN 0
           END IF
        END IF

        // Set insertion point
        rte_1.SelectText(llstart, lcstart, 0, 0)
        // In case it's a new line, get new line length
        // Can't do this until the ins pt is in the line
        ll_chars = rte_1.LineLength( )

        // Select word, if any
        li_rtn = rte_1.SelectTextWord()
    LOOP

    // Add code here to process the word (for example,
    // passing the word to a spelling checker)
```

See also

SelectedText
SelectText
SelectTextAll
SelectTextLine

Send

Description

Sends a message to a window so that it is executed immediately.

Syntax

Send (*handle, message#, lowword, long*)

Argument	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
lowword	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. If any argument's value is NULL, Send returns NULL.

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 sent directly to the object, bypassing 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.

Posting a message

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 CommandButton cb_OK:

```
Send(Handle(Parent), 273, 0, Handle(cb_OK))

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 properties 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 and DataStores

Syntax

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

Argument	Description
controlname	The name of the graph for which you want the number of series, or the name of the DataWindow control or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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. If any argument's value is NULL, SeriesCount returns NULL.

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 and DataStores

Syntax

controlname.**SeriesName** ({ *graphcontrol*, } *seriesnumber*)

Argument	Description
controlname	The name of the graph in which you want the name of a series, or the name of the DataWindow or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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 (""). If any argument's value is NULL, SeriesName returns NULL.

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.

Obsolete function
SetActionCode is obsolete and will be discontinued in the near future. You should replace all use of SetActionCode as soon as possible. To return a value, include a RETURN statement in the event script using the return codes documented for that event.

Applies to

DataWindow controls and child DataWindows

Syntax

dwcontrol.**SetActionCode** (*code*)

Argument	Description
dwcontrol	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. If any argument's value is NULL, SetActionCode returns NULL.

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.

Should be last statement
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.

Examples

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)
```

SetAlignment

Description

Sets the alignment of the selected paragraphs in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**SetAlignment** (*align*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to set the alignment of selected paragraphs
align	A value of the Alignment enumerated data type specifying how to align the paragraphs. Values are: ♦ Left! — Align each line at the left margin ♦ Right! — Align each line at the right margin ♦ Center! — Center the text between the left and right margins ♦ Justify! — Justify the paragraphs

Return value

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

Examples

This example sets the alignment of the selected paragraphs in the RichTextEdit control rte_1:

```
integer li_success
li_success = rte_1.SetAlignment(Right!)
```

See also

GetAlignment
GetSpacing
GetTextStyle
SetSpacing
SetTextStyle

SetArgElement

Description

Sets the value in the specified argument element.

Applies to

Window ActiveX controls

Syntax

activexcontrol.**SetArgElement** (*index*, *argument*)

Argument	Description
activexcontrol	Identifier for the instance of the PowerBuilder window ActiveX control. When used in HTML, this is the NAME attribute of the object element. When used in other environments, this references the control that contains the PowerBuilder window ActiveX
index	Integer specifying argument placement
argument	Any specifying the argument value

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function before calling InvokePBFunction or TriggerPBEvent to specify an argument for the passed function.

JavaScript scripts must use this function to specify function and event arguments. VBScript scripts can either use this function or specify the arguments array directly.

Examples

This JavaScript example calls the SetArgElement function:

```
function triggerEvent(f) {
   var retcd;
   var rc;
   var numargs;
   var theEvent;
   var theArg;
   retcd = 0;
   numargs = 1;
   theArg = f.textToPB.value;
   PBRX1.SetArgElement(1, theArg);
   theEvent = "ue_args";
   retcd = PBRX1.TriggerPBEvent(theEvent, numargs);
...
```

See also

GetArgElement
GetLastReturn
InvokePBFunction
TriggerPBEvent

SetAutomationLocale

Description

Sets the language to be used in automation programming for an OLE object. Call SetAutomationLocale if you have programmed automation commands in a language other than the user's locale.

Applies to

OLE objects

Syntax

olename.**SetAutomationLocale** (*language, sortorder*)

Argument	Description
olename	The name of the object for which you want to set the automation locale
language	A value of the LanguageID enumerated data type specifying the language you have used for automation commands. The OLE server must have function and property names defined in the language you specify. Some values of LanguageID are: ♦ LanguageNeutral! — No language is assumed. Automation commands match the server's default command set. ♦ LanguageUserDefault! — The language locale is taken from the user's settings in the International control panel. ♦ LanguageSystemDefault! — The language locale is taken from the version of Windows that is installed on the user's machine. You can also specify a language or dialect, such as LanguagePolish! or LanguagePortuguese_Brazilian! FOR INFO For the list of language-specific values for LanguageID, use the PowerBuilder Browser
sortorder	A value of the LanguageSortID enumerated data type specifying the sort order for the language. Values are: ♦ LanguageSortNative! — Use the traditional sort order of the selected language. ♦ LanguageSortUnicode! — Use the sort order defined for Unicode

Return value

Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage

For most situations, you will not need to call SetAutomationLocale. If an automation command fails, PowerBuilder will make additional attempts to execute it in other languages before it triggers the Error event. It attempts to execute the command using these languages:

1 The command as is (the command is in a language the server understands)

2 The current locale (if it is different from the user's default locale)

3 The user's default locale (LanguageUserDefault!)

4 The system's default locale (LanguageSystemDefault!)

5 English (LanguageEnglish!)

If PowerBuilder is successful in validating the name in any of the languages above, it resets the locale to the value that succeeded. While this may result in the wrong locale in ambiguous cases, it will probably simplify access to standard Microsoft Office products that ship with both localized and English function and property names.

If you specify a language with SetAutomationLocale, but the OLE server doesn't have function and property names in that language, your OLE automation commands will fail unless the above procedure finds a language that works. If you have called SetAutomationLocale, PowerBuilder's procedure for finding the correct language can reset it, as described in the previous paragraph.

Examples

This example sets the language to German for an OLEObject called oleobj_report:

```
oleobj_report.SetAutomationLocale(LanguageGerman!)
```

This example sets the language to German for an OLE control ole_1:

```
ole_1.Object.SetAutomationLocale(LanguageGerman!)
```

SetAutomationPointer

Description

Sets the automation pointer of an OLEObject object to the value of the automation pointer of another object.

Applies to

OLEObject

Syntax

oleobject.**SetAutomationPointer** (*object*)

Argument	Description
oleobject	The name of an OLEObject variable whose automation pointer you want to set. You cannot specify an OLEObject that is the Object property of an OLE control
object	The name of an OLEObject variable that contains the automation pointer you want to use to set the pointer value in *oleobject*

Return value

Integer. Returns 0 if it succeeds and -1 if the object does not contain a valid OLE automation pointer.

Usage

SetAutomationPointer assigns the underlying automation pointer used by OLE into a descendant of OLEObject.

Examples

This example creates an OLEObject variable and calls ConnectToNewObject to create a new Excel object and connect to it. It also creates an object of type oleobjectchild (which is a descendant of OLEObject) and sets the automation pointer of the descendant object to the value of the automation pointer in the OLEObject object. Then it sets a value in the worksheet using the descendent object, saves it to a different file, and destroys both objects:

```
OLEObject ole1
oleobjectchild oleChild
integer rs

ole1= CREATE OLEObject
rs = ole1.ConnectToNewObject("Excel.Application")

oleChild = CREATE oleobjectchild
rs = oleChild.SetAutomationPointer(ole1 )

IF ( rs = 0 ) THEN
    oleChild.workbooks.open("d:\temp\expenses.xls")
    oleChild.cells(1,1).value = 11111
    oleChild.activeworkbook.saveas( &
```

```
            "d:\temp\newexp.xls")
        oleChild.activeworkbook.close()
        oleChild.quit()
    END IF

    ole1.disconnectobject()
    DESTROY oleChild
    DESTROY ole1
```

See also

GetAutomationNativePointer
ReleaseAutomationNativePointer

SetAutomationTimeout

Description

Sets the number of milliseconds that a PowerBuilder client waits before canceling an OLE procedure call to the server.

Applies to

OLEObject objects

Syntax

oleobject.**SetAutomationTimeout** (*interval*)

Argument	Description
oleobject	The name of an OLEObject variable containing the object for which you want to set the timeout period
interval	A 32-bit signed long integer value (in milliseconds) specifying how long a PowerBuilder client waits before canceling a procedure call. The default value is 300,000 milliseconds (5 minutes). Specifying 0 or a negative value resets *interval* to the default value

Return value

Integer. Returns 0 if it succeeds and -1 if it fails.

Usage

This function passes the value of *interval* to PowerBuilder's implementation of the IMessageFilter interface and determines how long PowerBuilder tries to complete an OLE procedure call. The value applies only when PowerBuilder is the OLE client, not when PowerBuilder is the OLE server.

For most situations, you do not need to call SetAutomationTimeout. The default timeout period of five minutes is usually appropriate. Use SetAutomationTimeout to change the default timeout period if you expect a specific OLE request to take longer than five minutes.

If the timeout period expires, runtime error 1037 occurs. You may want to add code to handle this error, which is often the only indication of a hung server. Note that canceling a transaction often causes memory leaks on both the server and the operating system.

The value that you specify with SetAutomationTimeout applies to all OLE transactions in the current session, including calls that relate to other objects.

Examples

This example calls the ConnectToObject function to connect to an Excel worksheet and sets a timeout period of 900,000 milliseconds (15 minutes):

```
OLEObject ole1
integer rs
long interval

interval = 900000
```

```
ole1 = create OLEObject
rs = ole1.ConnectToObject("Excel.Application")
rs = ole1.SetAutomationTimeOut(interval)
```

See also

SetBorderStyle

Description

Sets the border style of a column in a DataWindow control or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetBorderStyle** (*column, borderstyle*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If any argument's value is NULL, SetBorderStyle returns NULL.

Examples

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

SetChanges

Description

Applies changes captured with GetChanges to a DataWindow or DataStore.

This function is used primarily in distributed applications.

Applies to

DataWindow controls and DataStore objects

Syntax

dwcontrol.**SetChanges** (*changeblob* {, *resolution* })

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore to which you want to apply the changes
changeblob	A read-only change blob created with GetChanges from which you want to apply changes
resolution (optional)	A value of the dwConflictResolution enumerated data type indicating how conflicts should be resolved: ♦ FailOnAnyConflict! (default) ♦ AllowPartialChanges!

Return value

Long. Returns one of the following values:

1 All changes were applied
2 A partial update was successful; conflicting changes were discarded
-1 Function failed
-2 There is a conflict between the state of the DataWindow change blob and the state of the DataWindow
-3 Column specifications do not match

Usage

Use this function in conjunction with GetChanges to synchronize two or more DataWindows or DataStores. GetChanges retrieves data buffers and status flags for changed rows in a DataWindow or DataStore and places this information in a blob. SetChanges then applies the contents of this blob to another DataWindow or DataStore.

In situations where a single DataStore on a server acts as the source for multiple target DataWindows (or DataStores) on different clients, you can use GetChanges in conjunction with GetStateStatus to determine the likely success of SetChanges. This allows you to avoid shipping a change blob across the wire when the SetChanges call will fail anyway because changes in the blob conflict with changes made previously by another client.

To determine the likely success of SetChanges, you need to:

1 Call the GetStateStatus function on the DataStore on which you want to do a SetChanges. GetStateStatus checks the state of the DataStore and makes the state information available in a reference argument called a **cookie**. The cookie is generally much smaller than a DataWindow change blob.

2 Send the cookie back to the client.

3 Call the GetChanges function on the DataWindow from which you want to apply changes, passing the cookie retrieved from GetStateStatus as a parameter. The return value from GetChanges indicates whether there are currently any potential conflicts between the state of the DataWindow blob and the state of the DataStore.

If the return value from GetChanges indicates that there are potential conflicts, you can then be certain that a subsequent call to SetChanges will fail if the FailOnAnyConflict! argument is specified. However, if the return value from GetChanges indicates no conflicts, the call to SetChanges may *still* fail, because the state of the DataStore may have changed since you called GetStateStatus and GetChanges. For example, if another client session has called SetChanges or some other processing has been executed that altered the state of the DataStore since you retrieved the cookie, SetChanges will fail.

Examples

The following example is a script for a remote object function. The script uses SetChanges to apply changes made to a DataWindow control on a client to a DataStore on a server. The changes made on the client are contained in a change blob that is passed as an argument to the function. After applying changes to the DataStore, the server updates the database:

```
// Instance variable:datastore ids_datastore
// Function argument: blob ablb_data
long ll_rv

ids_datastore.SetChanges(ablb_data)
ll_rv = ids_datastore.Update()

IF ll_rv > 0 THEN
   COMMIT;
ELSE
   ROLLBACK;
END IF
RETURN ll_rv
```

See also

GetChanges
GetFullState
GetStateStatus
SetFullState

SetColumn

Sets column information for a DataWindow, child DataWindow, or ListView control.

To set	Use
The number of the current column in a DataWindow or child DataWindow control	Syntax 1
The properties of a specified column in a ListView control	Syntax 2

Syntax 1 For DataWindows and DataStores

Description

Sets the current column in a DataWindow control or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetColumn** (*column*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If any argument's value is NULL, SetColumn returns NULL.

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 its 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.

Examples

This statement makes the 15th column in dw_Employee the current column:

```
dw_Employee.SetColumn(15)
```

See also

GetColumn
GetRow
SetRow

Syntax 2

For ListView controls

Description

Sets the properties of a column in a ListView control.

Applies to

ListView controls

Syntax

listviewname.**SetColumn** (*index, label, alignment, width*)

Argument	Description
listviewname	The name of the ListView control for which you want to set column properties
index	The number of the column for which you want to set column properties
label	The label of the column for which you want to set column properties
alignment	A value of the Alignment enumerated data type specifying how to align the column. Values are: ♦ Left! — Align the column at the left margin ♦ Right! — Align the column at the right margin ♦ Center! — Center the column between the left and right margins ♦ Justify! — Not valid for the SetColumn function
width	The width of the column for which you want to set column properties

Return value

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

Usage

SetColumn is only used in report views.

Examples

This example sets the second column of a ListView:

```
lv_list.SetColumn(2 , "Order" , Center! , 800)
```

See also

SetItem

SetConnect

Description

Creates an instance of a remote object on a server.

This function applies to distributed applications only.

Obsolete function
SetConnect is obsolete and will be discontinued in the near future. You should replace all use of SetConnect as soon as possible. To create an instance of a remote object, use the CreateInstance function instead.

Applies to

Remote objects

Syntax

proxyobject.**SetConnect** (*connection*)

Argument	Description
proxyobject	The name of the proxy object for the remote object that you want to instantiate
connection	The name of the connection object for which a connection has been established

Return value

None.

Usage

Before calling the SetConnect function, the client application needs to create instances of both the proxy object and the connection object.

Examples

In this example, the client application instantiates the remote object that is associated with the po proxy object:

```
connection myconnect
po_remobject po
long result

myconnect = create connection
myconnect.driver = "WinSock"
myconnect.application = "dpbserv"
myconnect.location = "server01"
myconnect.ConnectToServer( )
IF myconnection.errcode <> 0 THEN
   MessageBox("Error","Connection failed")
END IF

po = create po_remobject
```

```
po.SetConnect(myconnect)
result = po.method1(parm)
```

See also CreateInstance

SetData

Description

Sets data in the OLE server associated with an OLE control using Uniform Data Transfer.

Applies to

OLE controls and OLE custom controls

Syntax

olename.**SetData** (*clipboardformat*, *data*)

Argument	Description
olename	The name of the OLE or custom control associated with the OLE server to which you want to transfer data
clipboardformat	The format of the data. You can specify a standard format with a value of the ClipboardFormat enumerated data type. You can specify a nonstandard format as a string. Values for ClipboardFormat are: ClipFormatBitmap! ClipFormatDIB! ClipFormatDIF! ClipFormatEnhMetafile! ClipFormatHdrop! ClipFormatLocale! ClipFormatMetafilePict! ClipFormatOEMText! ClipFormatPalette! ClipFormatPenData! ClipFormatRIFF! ClipFormatSYLK! ClipFormatText! ClipFormatTIFF! ClipFormatUnicodeText! ClipFormatWave! If *clipboardformat* is an empty string or a NULL value, SetData transfers the data with the format ClipFormatText!
data	A string or blob whose value is the data you want to transfer

Return value

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

Usage

SetData will return an error if you specify a clipboard format that the OLE server doesn't support. See the documentation for the OLE server to find out what formats it supports.

SetData operates via Uniform Data Transfer, a mechanism defined by Microsoft for exchanging data with container applications. PowerBuilder enables data transfer via a global handle. The OLE server must also support data transfer via a global handle. If it does not, you cannot transfer data to or from that server.

Examples

For an example of moving data between two OLE controls (a Microsoft Word table and a Microsoft Graph), see GetData.

See also

GetData

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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

SetDataDDE (*string* {, *applname*, *topic*, *item* })

Argument	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

If any argument's value is NULL, SetDataDDE returns NULL.

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 and DataStores

Syntax

controlname.**SetDataPieExplode** ({ *graphcontrol*, } *seriesnumber*, *datapoint*, *percentage*)

Argument	Description
controlname	The name of the graph in which you want to explode a pie slice, or the name of the DataWindow or DataStore containing the graph.
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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. If any argument's value is NULL, SetDataPieExplode returns NULL.

Usage

If the graph is not a pie graph, the function has no effect.

Examples

This example explodes the pie slice under the pointer to 50% when the user double-clicks within the graph. The code checks the property 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

Specifies the appearance of a data point in a graph. The data point's series has appearance settings that you can override with SetDataStyle.

To	Use
Set the data point's colors	Syntax 1
Set the line style and width for the data point	Syntax 2
Set the fill pattern or symbol for the data point	Syntax 3

Syntax 1

For setting a data point's colors

Description

Specifies the colors of a data point in a graph.

Applies to

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

Syntax

controlname.**SetDataStyle** ({ *graphcontrol*, } *seriesnumber*, *datapointnumber*, *colortype*, *color*)

Argument	Description
controlname	The name of the graph in which you want to set the color of a data point, or the DataWindow or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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

Argument	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetDataStyle returns NULL.

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, 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)
```

See also

GetDataStyle
GetSeriesStyle
ResetDataColors
SeriesName
SetSeriesStyle

Syntax 2

For the line associated with a data point

Description

Specifies the style and width of a data point's line in a graph.

Applies to

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

Syntax

controlname.**SetDataStyle** ({ *graphcontrol*, } *seriesnumber*, *datapointnumber*, *linestyle*, *linewidth*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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

Argument	Description
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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetDataStyle returns NULL.

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.

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 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)
```

See also

GetDataStyle
GetSeriesStyle
SeriesName
SetSeriesStyle

Syntax 3

For the fill pattern and symbol of a data point

Description

Specifies the fill pattern and symbol for a data point in a graph.

Applies to

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

Syntax

controlname.**SetDataStyle** ({ *graphcontrol*, } *seriesnumber*, *datapointnumber*, *enumvalue*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control and DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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

Argument	Description
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! SymbolSolidCircle! SymbolSolidDiamond! SymbolPlus! SymbolHollowDownArrow! SymbolSolidBox!

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetDataStyle returns NULL.

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.

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 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
GetSeriesStyle
SeriesName
SetSeriesStyle

SetDetailHeight

Description

Sets the height of each row in the specified range to the specified value.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetDetailHeight** (*startrow, endrow, height*)

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore 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. If any argument's value is NULL, SetDetailHeight returns NULL.

Usage

Call SetDetailHeight in a script to vary the amount of space assigned to rows in a DataWindow control or DataStore. 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 property 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)
```

SetDropHighlight

Description

Highlights the specified item as the drop target.

Applies to

TreeView controls

Syntax

treeviewname.**SetDropHighlight** (*itemhandle*)

Argument	Description
treeviewname	The TreeView control in which you want to highlight an item as the target of a drag-and-drop operation
itemhandle	The handle of the item you want to highlight as the target in a drag-and-drop operation

Return value

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

Usage

Use in a drag operation to specify a drop target.

Examples

This example uses the TreeView Clicked event to set the current TreeView item as the drop target:

```
handle = tv_list.FindItem(CurrentTreeItem!,0)
tv_list.SetDropHighlight(handle)
```

See also

FindItem
SetItem

SetDynamicParm

Description

Specifies a value for an input parameter in the DynamicDescriptionArea that will be used in an SQL OPEN or EXECUTE statement.

Only for Format 4 dynamic SQL
Use this function only in conjunction with Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**SetDynamicParm** (*index*, *value*)

Argument	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. If any argument's value is NULL, SetDynamicParm returns NULL.

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
Using dynamic SQL
OPEN Cursor

SetFilter

Description

Specifies filter criteria for a DataWindow control or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetFilter** (*format*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 or DataStore, 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.

To let users specify their own filter expression for a DataWindow control, you can pass a null string to the SetFilter function. PowerBuilder displays its Specify Filter dialog box with the filter expression blank. Then you can call Filter to apply the user's filter expression to the DataWindow. You cannot pass a null string to the SetFilter function for a DataStore object.

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. The final statement calls Filter to apply the filter:

```
string DWfilter2
DWfilter2 = "cust_qty > 100 and cust_code >30"
dw_Employee.SetFilter(DWfilter2)
dw_Employee.Filter( )
```

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("#1="+ String( min_qty) &
+ " and #2=" + String(max_qty))
```

The following example sets the filter expression to null, which causes PowerBuilder to display the Specify Filter dialog box. Then it calls Filter, which applies the filter expression the user specified:

```
string null_str
SetNull(null_str)
dw_main.SetFilter(null_str)
dw_main.Filter()
```

See also

Filter

SetFirstVisible

Description

Sets the specified item as the first visible item in a TreeView control.

Applies to

TreeView controls

Syntax

treeviewname.**SetFirstVisible** (*itemhandle*)

Argument	Description
treeviewname	The TreeView control in which you want to identify an item as the first visible item
itemhandle	The handle of the item you are identifying as the first visible item in the TreeView control

Return value

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

Usage

Use to give focus to the TreeView item specified by the *itemhandle* and scroll it to the top of the TreeView control (or as close to the top as the item list will allow; if the item is the last item in a TreeView control, for example, it cannot scroll to the top of the control).

Examples

This example sets the current TreeView item as the first item visible in a TreeView control:

```
long ll_tvi
int li_tvret

ll_tvi = tv_list.FindItem(CurrentTreeItem! , 0)

li_tvret = tv_list.SetFirstVisible(ll_tvi)
IF li_tvret = -1 THEN
   MessageBox("Warning!" , "Didn't Work")
END IF
```

See also

FindItem
SetItem

SetFocus

Description

Sets the focus on the specified object or control.

Applies to

Any object

Syntax

objectname.**SetFocus** ( )

Argument	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. If *objectname* is NULL, SetFocus returns NULL.

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.

Examples

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

SetItem
SetState
SetTop

SetFormat

Description

Specifies a display format for a column in a DataWindow control or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetFormat** (*column*, *format*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If any argument's value is NULL, SetFormat returns NULL. The return value is usually not used.

Usage

For information on valid display formats for different data types, see the *PowerBuilder User's Guide* or the Display Formats dialog box 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.

An Editmask edit style supersedes any display format applied to the column. When the column has an EditMask edit style, calling SetFormat will have no effect.

Examples

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

SetFullState

Description

Applies the contents of a DataWindow blob retrieved by GetFullState to a DataWindow or DataStore.

This function is used primarily in distributed applications.

Applies to

DataWindow controls and DataStore objects

Syntax

dwcontrol.**SetFullState** (*dwasblob*)

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore to which you want to apply the blob
dwasblob	A blob that contains the state information you want to apply to the DataWindow control or DataStore

Return value

Long. Returns -1 if an error occurs and one of the following values if it succeeds:

1 DataWindow objects match; old data and state overwritten
2 DataWindow objects do not match; old object, data, and state replaced
3 No DataWindow object associated with DataWindow control or DataStore; new one assigned from DataWindow blob

Usage

GetFullState retrieves the entire state of a DataWindow or DataStore into a blob, including the DataWindow object specification, the data buffers, and the status flags. When you use SetFullState to apply the blob created by GetFullState to another DataWindow, the target DataWindow has enough information to recreate the source DataWindow.

Because the blob created by GetFullState contains the DataWindow object specification, a subsequent call to SetFullState will overwrite the DataWindow object for the target DataWindow control or DataStore. If the target of SetFullState does not have a DataWindow object associated with it, the blob will assign one. In this case, SetFullState has the effect of setting the DataObject property for the target.

When you use GetFullState and SetFullState to synchronize a DataWindow control on a client with a DataStore on a server, you need to make sure that the DataWindow object for the DataStore contains the presentation style you want to display on the client.

Examples

These statements in a distributed client application call a remote object function that retrieves database information into a DataStore and puts the contents of the DataStore into a blob by using GetFullState. After the server passes the blob back to the client, the client uses SetFullState to apply the blob to a DataWindow control:

```
// Global variable:connection myconnect
// Instance variable: uo_employee iuo_employee

blob lblb_data
long ll_rv

myconnect.CreateInstance(iuo_employee)
iuo_employee.RetrieveData(lblb_data)

ll_rv = dw_empdata.SetFullState(lblb_data)

if ll_rv = -1 then
   MessageBox("Error", "SetFullState call failed!")
end if
```

See also

GetChanges
GetFullState
GetStateStatus
SetChanges

SetItem

Sets the value of an item in a list or a DataWindow row and column.

To set the values of an item in	Use
A DataWindow control or DataStore	Syntax 1
A ListView control	Syntax 2
A ListView control column	Syntax 3
A TreeView control	Syntax 4

Syntax 1

For DataWindows, DataStores, and child DataWindows

Description

Sets the value of a row and column in a DataWindow control or DataStore to the specified value.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetItem** (*row, column, value*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 or DataStore 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 use a return code that rejects the data the user entered but allows the focus to change (return code of 2 in the script of the ItemChanged event or return code of 3 in the ItemError event), you can call SetItem to put valid data in the row and column.

Using SetItem to correct user input

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

SetNull(null_num)
```

```
// Special processing for column 2
IF dwo.ID = 2 THEN
   // If user entered nothing (""), set to null
   IF data = "" THEN
      This.SetItem(row, dwo.ID, null_num)
      RETURN 2
   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 return code of 3 causes the data in the edit control to be rejected because the script has provided a valid value:

```
string snum, c
integer cnt

// Extract the digits from the user's data
FOR cnt = 1 to Len(data)
   c = Mid(data, cnt, 1) // Get character
   IF IsNumber(c) THEN snum = snum + c
NEXT
This.SetItem(row, dwo.ID, Long(snum))
RETURN 3
```

See also

GetItemDate
GetItemDateTime
GetItemNumber
GetItemString
GetItemTime
GetText
SetText

Syntax 2 For ListView controls

Description

Set the values for a ListView item.

Applies to

ListView controls

Syntax

listviewname.**SetItem** (*index, item*)

Argument	Description
listviewname	The ListView for which you are setting an item's state information
index	The index number of the item for which you are setting state information
item	The ListView item for which you are setting state information

Return value

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

Usage

To populate the columns or a report view for a ListView item, use Syntax 3.

Examples

This example uses SetItem to set the state picture for the selected ListView item:

```
listviewitem lvi_1

lv_list.GetItem(lv_list.SelectedIndex () , lvi_1)
lvi_1.StatePictureIndex = 2
lv_list.SetItem(lv_list.SelectedIndex () , lvi_1)
```

See also

GetItem

Syntax 3 For ListView controls

Description

Sets the values for a particular column.

Applies to

ListView control

Syntax

listviewname.**SetItem** (*index, column, label*)

Argument	Description
listviewname	The ListView control for which you are setting an item's column information
index	The index number of the item for which you are setting column information
column	The number of the column you are populating for the specified ListView item
label	The name of the item with which you are populating the specified column

Return value

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

Usage

To specify the information for a ListView item, use Syntax 2.

Examples

This example populates three columns of a report view for three existing ListView items:

```
listviewitem l_lvi
integer li_count

lv_list.SetColumn(1 , "Composition", Left! , 860)
lv_list.SetColumn(2 , " Album", Left! , 610)
lv_list.SetColumn(3 , " Artist", Left! , 710)

lv_list.SetItem(1 , 1 , "St. Thomas")
lv_list.SetItem(1 , 2 , "The Bridge")
lv_list.SetItem(1 , 3 , "Sonny Rollins")

lv_list.SetItem(2 , 1 , "So What")
lv_list.SetItem(2 , 2 , "Kind of Blue")
lv_list.SetItem(2 , 3 , "Miles Davis")

lv_list.SetItem(3 , 1 , "Goodbye, Porkpie Hat")
lv_list.SetItem(3 , 2 , "Mingus-Ah-Um")
lv_list.SetItem(3 , 3 , "Charles Mingus")
```

See also

GetItem

Syntax 4

For TreeView controls

Description

Sets the data associated with a specified item.

Applies to

TreeView controls

Syntax

treeviewname.**SetItem** (*itemhandle, item*)

Argument	Description
treeviewname	The name of the TreeView control in which you want to set the data for a specific item
itemhandle	The handle associated with the item you want to change
item	The TreeView item you want to change

Return value

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

Usage

Typically, you would call GetItem first, edit the data, and then call SetItem to reflect your changes in the TreeView control.

Examples

This example uses the ItemExpanding event to change the picture index and selected picture index of the current TreeView item:

```
treeviewitem l_tvi
long ll_tvi

ll_tvi = tv_list.FindItem(CurrentTreeItem! , 0)
tv_list.GetItem(ll_tvi , l_tvi)
l_tvi.PictureIndex = 5
l_tvi.SelectedPictureIndex = 5

tv_list.SetItem( ll_tvi, l_tvi )
```

See also

GetItem

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, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetItemStatus** (*row, column, dwbuffer, status*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 INFO For definitions of the values of dwItemStatus, see GetItemStatus

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetItemStatus returns NULL.

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!**	**New Modified!**	**Data Modified!**	**Not Modified!**
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, 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!.

Resetting status for the whole DataWindow object

To reset the update status of the entire DataWindow object, use the ResetUpdate function. This sets all status flags to NotModified! except for New! status flags which remain unchanged.

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

SetLevelPictures

Description

Sets the picture indexes for all items at a particular level.

Applies to

TreeView controls

Syntax

treeviewname.**SetLevelPictures** (*level, pictureindex, selectedpictureindex, statepictureindex, overlaypictureindex*)

Argument	Description
treeviewname	The TreeView control in which you want to set the pictures for a given TreeView level
level	The TreeView level for which you are setting the picture indexes
pictureindex	An index from the regular picture list specifying the picture to be displayed when the item is not selected
selectedpictureindex	An index from the regular picture list specifying the picture to be displayed when the item is selected
statepictureindex	An index from the state picture list specifying the picture to be displayed to the left of the regular picture
overlaypictureindex	An index from the overlay picture list specifying the picture to be displayed on top of the regular picture

Return value

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

Usage

To set pictures for individual items, call GetItem, set the picture properties, and call SetItem to copy the changes to the TreeView.

You must specify a value for all four indexes. To display nothing, specify 0.

Examples

This example sets the pictures for TreeView level 3, then inserts two new TreeView items:

```
long ll_tvi, ll_child, ll_child2
int li_pict, li_level
treeviewitem l_tvi

li_level = 6

tv_list.SetLevelPictures( 3, li_level, li_level, &
   li_level, li_level)

ll_tvi = tv_list.FindItem(RootTreeItem! , 0)
ll_child = tv_list.InsertItemLast(ll_tvi, "Walton",2)
ll_child2 = tv_list.InsertItemLast(ll_child, &
```

```
            "Spitfire Suite", li_level)

        tv_list.ExpandItem(ll_child)
        tv_list.SetFirstVisible(ll_child)
```

See also AddPicture

SetLibraryList

Description

Changes the files in the library search path of the application.

Applies to

Application object

Syntax

applicationname.**SetLibraryList** (*filelist*)

Argument	Description
applicationname	The name of the application object for which you want to change the library search path
filelist	A comma-separated list of filenames. 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. If an error occurs, it returns:

- -1 The application is being run from PowerBuilder, rather than from a standalone executable.
- -2 A currently instantiated object is in a library that is not on the new list.

If any argument's value is NULL, SetLibraryList returns NULL.

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 specify a different list of library files from those specified in the executable with SetLibraryList. Calling SetLibraryList replaces the list of library files specified in the executable with a new list of files. For example, you might use SetLibraryList to configure the library list for an application containing many subsystems.

SetLibraryList should only be called in your application's Open event script. Otherwise, it may cause your application to crash.

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 your application in the PowerBuilder development environment, this function has no effect.

Examples

This example specifies different files in the library search path based on the selected application subsystem:

```
CHOOSE CASE configuration
   CASE "Config1"
      myapp.SetLibraryList("lib1.pbd, lib2.pbd, &
         lib5.pbd")
   CASE "Config2"
      myapp.SetLibraryList("lib1.pbd, lib3.pbd, &
         lib4.pbd")
END CHOOSE
```

SetMask

Description

Sets the edit mask and edit mask data type for an EditMask control.

Applies to

EditMask controls

Syntax

editmaskname.**SetMask** (*maskdatatype*, *mask*)

Argument	Description
editmaskname	The name of the EditMask for which you want to specify the edit mask
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. If any argument's value is NULL, SetMask returns NULL.

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 box for an EditMask control in the Window painter. A listbox in the dialog box 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 a 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 four 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*)

Argument	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. If any argument's value is NULL, SetMicroHelp returns NULL.

Usage

The Tag property 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 property'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 the menu item. You can use SetMicroHelp in the script for a menu item'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 except for an array, structure, or autoinstantiated object.

Syntax

SetNull (*anyvariable*)

Argument	Description
anyvariable	The variable you want to set to NULL

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetNull returns NULL.

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.

If you assign a value to a variable whose data type is Any and then set the variable to NULL, the data type of the NULL value is still the data type of the assigned value. You cannot untype an Any variable with the SetNull function.

Examples

This statement sets the variable Salary to NULL:

```
SetNull(Salary)
```

See also

IsNull

SetOverlayPicture

Description

Puts an image in the control's image list into an overlay image list.

Applies to

ListView and TreeView controls

Syntax

controlname.**SetOverlayPicture** (*overlayindex, imageindex*)

Argument	Description
controlname	The name of the ListView or TreeView control to which you want to add an overlay image
overlayindex	The index number of the overlay picture in the overlay image list. The overlay image list is a 1-based array. *Overlayindex* must be 1 (for the first image), a previously designated index (replacing an image), or 1 greater than the current largest index (adding another image). SetOverlayPicture fails if you specify an index that creates gaps in the array
imageindex	The index number of an image in the control's main image list. For ListViews, both the large and small pictures at that index become overlay images. The image is still available for use as an item's main image

Return value

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

Usage

An overlay picture must have the same height and width as the picture it is used to overlay.

The color specified in the SetPictureMask property when the picture is inserted becomes transparent when the picture is used as an overlay, allowing part of the original image to be visible beneath the overlay.

The overlay list acts as a pointer back to the source image in the regular picture lists. If you delete an image that is also used in the overlay list, the displayed overlay pictures are affected too.

Examples

This example designates overlay images in a ListView control. The same picture is used for large and small images:

```
// Set up the overlay images
integer index
index = lv_1.AddLargePicture("shortcut.ico")
index = lv_1.AddSmallPicture("shortcut.ico")
lv_1.SetOverlayPicture(1, index)
index = lv_1.AddLargePicture("not.ico")
```

```
index = lv_1.AddSmallPicture("not.ico")
lv_1.SetOverlayPicture(2, index)

// Assign the second overlay image to the first item
listviewitem lvi
integer i
i = lv_1.GetItem(1, lvi)
lvi.OverlayPictureIndex = 2
i = lv_1.SetItem(1, lvi)
```

This example designates the first picture in the TreeView's main image list as the first overlay picture. The picture was added to the main image list on the TreeView's property sheet:

```
tv_list.SetOverlayPicture(1, 1)
```

This code in the TreeView's Clicked event assigns the overlay image to the clicked item:

```
treeviewitem tvi
tv_list.GetItem(handle, tvi)
tvi.OverlayPictureIndex = 1
tv_list.SetItem(handle, tvi)
```

SetParagraphSetting

Description

Sets the size of the indentation, left margin, or right margin of the paragraph containing the insertion point in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtecontrol.**SetParagraphSetting** (*whichsetting, value*)

Argument	Description
rtecontrol	The name of the control for which you want paragraph information
whichsetting	A value of the ParagraphSetting enumerated data type specifying the setting you want to change. Values are: ♦ Indent! — Returns the indentation of the paragraph ♦ LeftMargin! — Returns the left margin of the paragraph ♦ RightMargin! — Returns the right margin of the paragraph
value	A long whose value is the width of the margin or indent in units of 1000ths of an inch. For example, a value of 500 specifies a width of half an inch

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument is NULL, it returns NULL.

Usage

Each paragraph has indentation, left margin, and right margin settings. To set all three for the current paragraph, call SetParagraphSetting three times.

Examples

This example sets the indentation setting for the current paragraph to a quarter inch:

```
ll_indent = rte_1.SetParagraphSetting(Indent!, 250)
```

This example sets the left margin for the current paragraph to an inch:

```
rte_1.SetParagraphSetting(LeftMargin!, 1000)
```

See also

GetParagraphSetting
SetAlignment
SetSpacing
SetTextColor
SetTextStyle

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*)

Argument	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. If any argument's value is NULL, SetPicture returns NULL.

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.

Examples

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*)

Argument	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. If *type* is NULL, SetPointer returns NULL.

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.

Restoring the arrow pointer
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

Specifies the front-to-back position of a control in a window, a window, or an object within a DataWindow.

To	Use
Specify the front-to-back position of a control in a window *or* Specify that a window should always display on top of other windows	Syntax 1 Syntax 1
Move an object in a DataWindow to another band or to specify its front-to-back position within a band	Syntax 2

Syntax 1

For positioning windows and controls in windows

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.

Applies to

A control within a window or a window

Syntax

objectname.**SetPosition** (*position* {, *precedingobject* })

Argument	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

Argument	Description
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

Integer. Returns 1 when it succeeds and -1 if an error occurs. If any argument's value is NULL, Set Position returns NULL.

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 For positioning objects within a DataWindow

Description

Moves an object within the DataWindow to another band or changes the front-to-back order of objects within a band.

Applies to

DataWindow controls and DataStores

Syntax

dwcontrol.**SetPosition** (*objectname, band, bringtofront*)

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore 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
band	The name of the band or layer in which you want to position *objectname*. Layer names are background and foreground. Band names are detail, header, footer, summary, header.#, and trailer.#. # 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

Integer. Returns 1 when it succeeds and -1 if an error occurs. If any argument's value is NULL, SetPosition returns NULL.

Examples

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*)

Argument	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. If any argument's value is NULL, SetProfileString returns NULL.

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.

Accessing the profile file
SetProfileString uses Window profile calls to write data to the profile file. Consequently it does not control when the profile file is written and closed. If you try to read data from the profile file immediately after calling SetProfileString, the file may still be open and you will receive incomplete or incorrect data.

To avoid this problem, you can use the PowerScript FileOpen, FileWrite, and FileClose functions to write data to the profile file instead of using SetProfileString. Or you can add some additional processing after the SetProfileString call so that the Windows calls have time to complete before you try to read from the profile file.

Windows NT and Windows 95
On 32-bit systems, SetProfileString can also be used to obtain configuration settings from the Windows system registry. For information on how to use the system registry, see the discussion of initialization files and the Windows registry in *Application Techniques*.

Examples

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")
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
SetProfileString("HD:System Folder:Preferences:" + &
"My App:Profile Preferences", "Position", "Title", &
"MGR")
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
SetProfileString("/export/home/myapp/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 properties.

Applies to

Any object except a Menu

Syntax

objectname.**SetRedraw** (*boolean*)

Argument	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 properties ♦ FALSE — Do not redraw after each change

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If *boolean* is NULL, SetRedraw returns NULL.

Usage

By default, PowerBuilder redraws a control after each change to properties 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 properties of an object or control.

If the window is not visible, SetRedraw will fail.

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 property 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

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.

To	Use
Make a single DDE request of a server application (a cold link)	Syntax 1
Make a DDE request of a server application when you have established a warm link by opening a channel	Syntax 2

Syntax 1

For single DDE requests

Description

Asks a DDE server application to accept data to be stored in the specified location without requiring an open channel. This syntax is appropriate when you will make only one or two requests of the server.

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

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

SetRemote (*location, value, applname, topicname*)

Argument	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

Integer. Returns 1 if it succeeds and a negative integer if an error occurs. Values are:

-1 Link was not started
-2 Request denied

If any argument's value is NULL, SetRemote returns NULL.

Usage

When using DDE, your PowerBuilder application must have an open window, which will be the client window. For this syntax, the active window is the DDE client window.

FOR INFO For more information about DDE channels and warm and cold links, see the ExecRemote function.

Examples

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")
```

See also

ExecRemote
GetRemote
OpenChannel

Syntax 2

For DDE requests via an open channel

Description

Asks a DDE server application to accept data to be stored in the specified location when you have already established a warm link by opening a channel to the server. 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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

SetRemote (*location, value, handle* {, *windowhandle* })

Argument	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

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 this syntax, you can specify a client window other than the active window with the *windowhandle* argument.

Before using this syntax of SetRemote, call OpenChannel to establish a DDE channel.

FOR INFO For more information about DDE channels and warm and cold links, see the ExecRemote function.

Examples

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 or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetRow** (*row*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If any argument's value is NULL, SetRow returns NULL.

Usage

SetRow moves the cursor to the current row but does not scroll the DataWindow control or DataStore.

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

dwcontrol.**SetRowFocusIndicator** (*focusindicator* {, *xlocation* {, *ylocation* } })

Argument	Description
dwcontrol	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 ♦ 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. If any argument's value is NULL, SetRowFocusIndicator returns NULL.

Usage

Sets the current row indicator in *dwcontrol* 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

Specifies the appearance of a series in a graph. There are several syntaxes, depending on what settings you want to change.

To	Use
Set the series' colors	Syntax 1
Set the line style and width	Syntax 2
Set the fill pattern or symbol for the series	Syntax 3
Specify that the series is an overlay	Syntax 4

Syntax 1 For setting a series' colors

Description

Specifies the colors of a series in a graph.

Applies to

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

Syntax

controlname.**SetSeriesStyle** ({ *graphcontrol*, } *seriesname*, *colortype*, *color*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control or DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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

Argument	Description
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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetSeriesStyle returns NULL.

Usage

Data points in a series can have their own style settings. Settings made with SetDataStyle set the style of individual data points and override series settings.

The graph stores style information for properties that don't apply to the current graph type. For example, you can set the fill pattern in a two-dimensional line graph or the line style in a bar graph, but that fill pattern or line style will not be visible.

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
```

See also

GetDataStyle
GetSeriesStyle
SeriesName
SetDataStyle

Syntax 2 For lines in a graph

Description

Specifies the style and width of a series' lines in a graph.

Applies to

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

Syntax

controlname.**SetSeriesStyle** ({ *graphcontrol*, } *seriesname*, *linestyle*, *linewidth*)

Argument	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 or DataStore containing the graph

Argument	Description
graphcontrol (DataWindow control or DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetSeriesStyle returns NULL.

Usage

Data points in a series can have their own style settings. Settings made with SetDataStyle set the style of individual data points and override series settings.

The graph stores style information for properties that don't apply to the current graph type. For example, you can set the fill pattern in a 2-dimensional line graph or the line style in a bar graph, but that fill pattern or line style will not be visible.

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 line style and width for the series named Costs in the graph gr_product_data:

```
gr_product_data.SetSeriesStyle("Costs", &
Dot!, 5)
```

See also

GetDataStyle
GetSeriesStyle
SeriesName
SetDataStyle

Syntax 3 For the fill pattern and symbols in a graph

Description

Specifies the fill pattern and symbol for data markers in a series.

Applies to

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

Syntax

controlname.**SetSeriesStyle** ({ *graphcontrol*, } *seriesname*, *enumvalue*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control or DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore 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

Argument	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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetSeriesStyle returns NULL.

Usage

Data points in a series can have their own style settings. Settings made with SetDataStyle set the style of individual data points and override series settings.

The graph stores style information for properties that don't apply to the current graph type. For example, you can set the fill pattern in a two-dimensional line graph or the line style in a bar graph, but that fill pattern or line style will not be visible.

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 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!)
```

See also

GetDataStyle
GetSeriesStyle
SeriesName
SetDataStyle

Syntax 4

For creating an overlay in a graph

Description

Specifies whether a series is an overlay, meaning that the series is represented by a line on top of another graph type.

Applies to

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

Syntax

controlname.**SetSeriesStyle** ({ *graphcontrol,* } *seriesname, overlaystyle*)

Argument	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 or DataStore containing the graph
graphcontrol (DataWindow control or DataStore only) (optional)	A string whose value is the name of the graph in the DataWindow control or DataStore in which you want to set the overlay status
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

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetSeriesStyle returns NULL.

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 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
GetSeriesStyle
SeriesName
SetDataStyle

SetSort

Description

Specifies sort criteria for a DataWindow control or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetSort** (*format*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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"

To let the user specify the sort criteria for a DataWindow control, you can pass a null string to the SetSort function. PowerBuilder displays the Specify Sort Columns dialog with the sort specifications blank. Then you can call Sort to apply the user's criteria. You cannot pass a null string to the SetSort function for a DataStore object.

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( )
```

The following example sets the sort criteria for dw_main to null, causing PowerBuilder to display the Specify Sort Columns dialog so that the user can specify sort criteria. The Sort function applies the criteria the user specifies:

```
string null_str
SetNull(null_str)
dw_main.SetSort(null_str)
dw_main.Sort( )
```

See also

Sort

SetSpacing

Description

Sets the line spacing for the selected paragraphs or the paragraph containing the insertion point in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**SetSpacing** (*spacing*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to set the line spacing
spacing	A value of the Spacing enumerated data type specifying the line spacing for the text. Values are: Spacing1! — Single spacing Spacing15! — One and a half line spacing Spacing2! — Double spacing

Return value

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

Usage

Because spacing is a setting for paragraphs, not individual lines, then if lines have wrapped, spacing will change for all the lines in all the paragraphs that are selected.

When you expand the line spacing, the extra space is added before the affected lines.

Examples

This example specifies double spacing for the selected paragraphs in the RichTextEdit rte_1:

```
rte_1.SetSpacing(Spacing2!)
```

This example specifies one and a half line spacing:

```
rte_1.SetSpacing(Spacing15!)
```

See also

SetTextColor
SetTextStyle

SetSQLPreview

Description

Specifies the SQL statement for a DataWindow control (or DataStore) that PowerBuilder is about to send to the database.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetSQLPreview** (*sqlsyntax*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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

Return value

Integer. Returns 1 if it succeeds and 0 if an error occurs. If any argument's value is NULL, SetSQLPreview returns NULL.

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.

Obsolete function

GetSQLPreview is obsolete and will be discontinued in the near future. You should replace all use of GetSQLPreview as soon as possible. The SQL syntax is available as an argument in the DBError and SQLPreview events.

When to call SetSQLPreview

Call this function *only* in the script for the SQLPreview event.

Examples

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 or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetSQLSelect** (*statement*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If any argument's value is NULL, SetSQLSelect returns NULL.

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 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* retrieval arguments and you want PowerBuilder to modify the update information for the DataWindow object:

```
dw_1.Modify("DataWindow.Table.Select='select...'")
```

Modify will not verify the SELECT statement or change the update information, making it faster but more susceptible to user error. Although you can use Modify when arguments are involved, it is not recommended because of the lack of checking.

Examples

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 property is set to TRUE.

Applies to

ListBox and PictureListBox controls

Syntax

listboxname.**SetState** (*index*, *state*)

Argument	Description
listboxname	The name of the ListBox or PictureListBox in which you want to set the state (highlighted or not highlighted) for an item. The MultiSelect property for the control 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. If any argument's value is NULL, SetState returns NULL.

Usage

When the MultiSelect property for the control 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:

```
IF 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

dwcontrol.**SetTabOrder** (*column*, *tabnumber*)

Argument	Description
dwcontrol	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. If any argument's value is NULL, SetTabOrder returns NULL.

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 or DataStore.

Applies to

DataWindow controls and DataStore objects

Syntax

dwcontrol.**SetText** (*text*)

Argument	Description
dwcontrol	The name of the DataWindow control or DataStore 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. If any argument's value is NULL, SetText returns NULL.

Usage

SetText only sets the value in the 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, use a return code that rejects the value in the edit control, avoiding further processing, and allow the focus to change. (Return 2 for ItemChanged and 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. The data argument holds the newly entered value:

```
integer negative
```

```
IF dwo.Name = "credit" THEN
   IF Integer(data) > 0 THEN
      // Convert to negative if it's positive
      negative = Integer(data) * -1
   END IF

   // Change the primary buffer value.
   This.SetItem(data, "credit", negative)

   // Change the value in the edit control
   This.SetText(String(negative))
   RETURN 2
END IF
```

See also

AcceptText
GetText

SetTextColor

Description

Sets the color of selected text in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

rtename.**SetTextColor** (*colornumber*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to set the color of selected text
colornumber	A long specifying the color of the selected text

Return value

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

Usage

For more information about calculating color values, see RGB.

Examples

This example sets the selected text in RichTextEdit rte_1 to dark red:

```
rte_1.SetTextColor(RGB(100, 0, 0))
```

See also

GetTextColor
RGB
SetTextStyle

SetTextStyle

Description

Specifies the text formatting for selected text in a RichTextEdit control. You can make the text bold, underlined, italic, and struck out. You can also make it either a subscript or superscript.

Applies to

RichTextEdit controls

Syntax

rtename.**SetTextStyle** (*bold*, *underline*, *subscript*, *superscript*, *italic*, *strikeout*)

Argument	Description
rtename	The name of the RichTextEdit control in which you want to specify formatting for selected text
bold	A boolean value specifying whether the selected text is bold
underline	A boolean value specifying whether the selected text is underlined
subscript	A boolean value specifying whether the selected text is a subscript
superscript	A boolean value specifying whether the selected text is a superscript. If both *subscript* and *superscript* are true, *subscript* takes precedence and the text is subscripted
italic	A boolean value specifying whether the selected text is italic
strikeout	A boolean value specifying whether the selected text is has a line drawn through it

Return value

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

Examples

This example makes selected text in the RichTextEdit rte_1 bold and italic:

```
rte_1.SetTextStyle(TRUE, FALSE, FALSE, FALSE, &
   TRUE, FALSE)
```

This example makes the selected text a subscript. It keeps other text formatting as it was:

```
rte_1.SetTextStyle(rte_1.GetTextStyle(Bold!), &
   rte_1.GetTextStyle(Underlined!), &
   TRUE, FALSE, &
   rte_1.GetTextStyle(Italic!), &
   rte_1.GetTextStyle(Strikeout!))
```

See also

GetTextStyle
SetSpacing
SetTextColor

SetToolbar

Description

Specifies the alignment, visibility, and title for the specified toolbar.

Applies to

MDI frame and sheet windows

Syntax

window.**SetToolbar** (*toolbarindex*, *visible* {, *alignment* {, *floatingtitle* } })

Argument	Description
window	The MDI frame or sheet to which the toolbar belongs
toolbarindex	An integer whose value is the index of the toolbar whose settings you want to change
visible	A boolean value specifying whether to make the toolbar visible. Values are: ♦ TRUE — Make the toolbar visible ♦ FALSE — Hide the toolbar
alignment (optional)	A value of the ToolbarAlignment enumerated data type specifying the alignment for the toolbar. Values are: ♦ AlignAtTop! — Dock the toolbar at the top of the frame. ♦ AlignAtLeft! — Dock the toolbar on the left side of the frame. ♦ AlignAtRight! — Dock the toolbar on the right side of the frame. ♦ AlignAtBottom! — Dock the toolbar at the bottom of the frame. ♦ Floating! — Float the toolbar. The floating toolbar has its own frame and miniature title bar
floatingtitle (optional)	A string whose value is the title for the toolbar when its alignment is Floating!

Return value

Integer. Returns 1 if it succeeds. SetToolbar returns -1 if there is no toolbar for the index you specify or if an error occurs. If any argument's value is NULL, returns NULL.

Usage

When you use SetToolbar to change the toolbar alignment from a docked position to Floating!, PowerBuilder uses the last known position information unless you also call SetToolbarPos to adjust the position.

The toolbars are not redrawn until the script ends, so setting the alignment with SetToolbar and the position with SetToolbarPos looks like a single change to the user.

Examples

This example allows the user to choose an alignment in a ListBox lb_position. The selected string is converted to a ToolbarAlignment enumerated value, which is used to change the alignment of toolbar index 1:

```
toolbaralignment tba_align

CHOOSE CASE lb_position.SelectedItem()

CASE "Top"
   tba_align = AlignAtTop!
CASE "Left"
   tba_align = AlignAtLeft!
CASE "Right"
   tba_align = AlignAtRight!
CASE "Bottom"
   tba_align = AlignAtBottom!
CASE "Floating"
   tba_align = Floating!
END CHOOSE

w_frame.SetToolbar(1, TRUE, tba_align)
```

In this example, the user clicks a radio button to choose an alignment. The radio button's Clicked event sets an instance variable of type ToolbarAlignment. Here the radio buttons are packaged as a custom visual user object. I_toolbaralign is an instance variable of the user object. This is the script for the Top radio button:

```
Parent.i_toolbaralign = AlignAtTop!
```

This script changes the toolbar alignment:

```
w_frame.SetToolbar(1, TRUE, &
   uo_toolbarpos.i_toolbaralign )
```

See also

GetToolbar
GetToolbarPos
SetToolbarPos

SetToolbarPos

Sets the position of the specified toolbar.

To set	Use
Docking position of a docked toolbar	Syntax 1
Coordinates and size of a floating toolbar	Syntax 2

Syntax 1

For docked toolbars

Description

Sets the position of a docked toolbar.

Applies to

MDI frame and sheet windows

Syntax

window.**SetToolbarPos** (*toolbarindex*, *dockrow*, *offset*, *insert*)

Argument	Description
window	The MDI frame or sheet to which the toolbar belongs
toolbarindex	An integer whose value is the index of the toolbar whose settings you want to change
dockrow	An integer whose value is the number of the docking row for the toolbar. Docking rows are numbered from left to right or top to bottom
offset	An integer whose value specifies the distance of the toolbar from the beginning of the docking row. For toolbars at the top or bottom, *offset* is measured from the left edge. For toolbars on the left or right, *offset* is measured from the top. If *replace* is TRUE, the *offset* you specify is adjusted so that the toolbar doesn't overlap others in the row. Specify an offset of 0 to position the toolbar ahead of other toolbars in *dockrow*

Argument	Description
insert	A boolean value specifying whether you want to insert the specified toolbar before the toolbars in *dockrow* causing them to move over or down a row, or you want to add *toolbarindex* to *dockrow*. Values are: ♦ TRUE — Move any toolbars already in *dockrow* or higher rows over or down a row so that the toolbar you are moving is the only toolbar in the row ♦ FALSE — Add the toolbar you are moving to *dockrow*. Its position in relation to other toolbars in the row is determined by *offset*

Return value

Integer. Returns 1 if it succeeds. SetToolbarPos returns -1 if there is no toolbar for the index you specify or if an error occurs. If any argument's value is NULL, returns NULL.

Usage

To find out whether the docked toolbar is at the top, bottom, left, or right edge of the window, call GetToolbar.

If the toolbar's alignment is floating, instead of docked, then values you specify with Syntax 1 of SetToolbarPos take effect when you change the alignment to a docked position with SetToolbar.

When *insert* is FALSE, to move the toolbar before other toolbars in *dockrow*, specify a value that is less than the offset for the existing toolbars. If there is already a toolbar at offset 1, then you can move the toolbar to the beginning of the row by setting *offset* to 0. If *offset* is equal to or greater than the offset of existing toolbars, but less than their end, the newly positioned toolbar will begin just after the existing one. Otherwise, the toolbar will be positioned at *offset*.

If the user drags the toolbar to a docked position, the new row and offset replace values set with SetToolbarPos.

Examples

This example docks toolbar 1 at the left, adding it to docking row 1:

```
w_frame.SetToolbar(1, TRUE, AlignAtLeft!)
w_frame.SetToolbarPos(1, 1, 1, FALSE)
```

This example docks toolbar 2 at the left, adding it to docking row 1. If the toolbars already in the dock extend past offset 250, then the offset of toolbar 2 is increased to accommodate them. Otherwise, it is positioned at offset 250:

```
w_frame.SetToolbar(2, TRUE, AlignAtLeft!)
w_frame.SetToolbarPos(2, 1, 250, FALSE)
```

This example docks toolbar 2 at the left in docking row 2. Any toolbar docked on the left in row 2 or higher is moved over a row:

```
w_frame.SetToolbar(1, TRUE, AlignAtLeft!)
w_frame.SetToolbarPos(1, 2, 1, TRUE)
```

See also

GetToolbar
GetToolbarPos
SetToolbar

Syntax 2

For floating toolbars

Description

Sets the position and size of a floating toolbar.

Applies to

MDI frame and sheet windows

Syntax

window.**SetToolbarPos** (*toolbarindex*, *x*, *y*, *width*, *height*)

Argument	Description
window	The MDI frame or sheet to which the toolbar belongs
toolbarindex	An integer whose value is the index of the toolbar whose settings you want to change
x	An integer whose value is the x coordinate of the floating toolbar
y	An integer whose value is the y coordinate of the floating toolbar
width	An integer whose value is the width of the floating toolbar
height	An integer whose value is the height of the floating toolbar

Return value

Integer. Returns 1 if it succeeds. SetToolbarPos returns -1 if there is no toolbar for the index you specify or if an error occurs. If any argument's value is NULL, returns NULL.

Usage

If the toolbar's alignment is a docked position, instead of floating, then values you specify with Syntax 2 of SetToolbarPos take effect when you change the alignment to floating in a script with SetToolbar.

If the user drags the toolbar to a floating position, the new position values replace values set with SetToolbarPos.

The floating toolbar is never too large or too small for the buttons. If you specify width and height values that are too small to accommodate the buttons, the width and height are adjusted to make room for the buttons. If both width and height are larger than needed, the height is reduced.

If you specify x and y coordinates that are outside the frame, the toolbar becomes inaccessible to the user.

Examples

This example displays toolbar 1 near the upper-left corner of the frame. An arbitrary width and height lets PowerBuilder size the toolbar as needed:

```
w_frame.SetToolbarPos(1, 10, 10, 400, 1)
w_frame.SetToolbar(1, TRUE, Floating!)
```

This example displays toolbar 2 close to the lower-right corner of the frame. GetToolbarPos gets the current width and height of the toolbar so that the toolbar stays the same size:

```
integer ix, iy, iw, ih

w_frame.GetToolbarPos(2, ix, iy, iw, ih)

w_frame.SetToolbarPos(2, &
    w_frame.WorkspaceWidth()-400, &
    w_frame.WorkspaceHeight()-400, &
    iw, ih)
w_frame.SetToolbar(2, TRUE, Floating!)
```

This example positions floating toolbar 2 just inside the lower-right corner of the MDI frame. GetToolbarPos gets the current width and height of the toolbar. These values and the height of the MicroHelp are used to calculate the x and y coordinates for the floating toolbar:

```
integer ix, iy, iw, ih

// Find out toolbar size
w_frame.GetToolbarPos(2, ix, iy, iw, ih)

// Set the position, taking the size into account
w_frame.SetToolbarPos(2, &
    w_frame.WorkspaceWidth( ) - iw, &
    w_frame.WorkspaceHeight( ) &
    - ih - w_frame.MDI_1.MicroHelpHeight, &
    iw, ih)
```

```
// Set the alignment to floating
w_frame.SetToolbar(2, TRUE, Floating!)
```

See also

GetToolbar
SetToolbar
SetToolbarPos

SetTop

Description

Scrolls a listbox control so that the specified item is the first visible item.

Applies to

ListBox and PictureListBox controls

Syntax

listboxname.**SetTop** (*index*)

Argument	Description
listboxname	The name of the ListBox or PictureListBox 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. If any argument's value is NULL, SetTop returns NULL.

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

SetTraceFileName

Description

Specifies the name of the trace file PowerBuilder will analyze when the BuildModel function is called.

Applies to

Profiling and TraceTree objects

Syntax

instancename.**SetTraceFileName** (*tracefilename*)

Argument	Description
instancename	Instance name of the Profiling or TraceTree object
tracefilename	A string that identifies the name of the trace file PowerBuilder will analyze

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileOpenError!—The file could not be opened
- FileInvalidFormatError!—The trace file is not in the correct format
- ModelExistsError!—A model has already been built

If an error occurs, the name is not set.

Usage

Use this function to specify the trace file PowerBuilder should analyze with the BuildModel function. You call the SetTraceFileName function before calling the BuildModel function.

Examples

This example provides the name of the trace file for which a performance analysis model is to be built:

```
Profiling lpro_model
String ls_line

lpro_model = CREATE Profiling

lpro_model.SetTraceFileName (filename)
ls_line = "CollectionTime = " + &
    String(lpro_model.CollectionTime ) + "~r~n" &
    + "Num Activities = " &
    + String(lpro_model.NumberOfActivities) + "~r~n"
```

```
lpro_model.BuildModel()
...
```

See also BuildModel

SetTrans

Description

Sets the values in the internal transaction object for a DataWindow control or DataStore to the values from the specified transaction object. The transaction object supplies connection settings, such as the database name.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetTrans** (*transaction*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol* to get values

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetTrans returns NULL.

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 internal transaction object for the DataWindow control or DataStore. 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.

DBMS connection settings 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.

Updating more than one table 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 or DataStore to use a programmer-specified transaction object. The transaction object provides the information necessary for communicating with the database.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetTransObject** (*transaction*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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 *dwcontrol*

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument's value is NULL, SetTransObject returns NULL.

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 or DataStore (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 DataStore) or if you disconnect and reconnect to a database, the connection between the DataWindow control (or DataStore) 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 internal transaction object for the DataWindow control or DataStore, 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.

FOR INFO For more information, see SetTrans.

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 or ORACLE 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

SetTransPool

Description

Sets up a pool of database transactions for an application. SetTransPool allows you to minimize the overhead associated with database connections and also limit the total number of database connections permitted.

Applies to

Application object

Syntax

applicationname.**SetTransPool** (*minimum*, *maximum*, *timeout*)

Argument	Description
applicationname	The name of the application object for which you want to establish a transaction pool
minimum	The minimum number of transactions to be kept open in the pool
maximum	The maximum number of transactions that can be open in the pool
timeout	The number of seconds to allow a request to wait for a connection in the transaction pool

Return value

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

Usage

Transaction pooling maximizes database throughput while also controlling the number of database connections that can be open at one time. By establishing a transaction pool before connecting to the database, an application can reuse connections made to the same data source.

When an application connects to a database without using transaction pooling, PowerBuilder physically terminates each database transaction for which a DISCONNECT statement is issued. When transaction pooling is in effect, PowerBuilder logically terminates the database connections and commits any database changes, but does not physically remove them. Instead, the database connections are kept open in the transaction pool so that they can be reused for other database operations.

Before reusing a connection in the transaction pool, PowerBuilder checks to see that the database parameters specified in the incoming connection request match those specified by one of the connections in the pool. A match occurs when both transaction objects specify the same DBMS, ServerName, LogID, LogPass, Database, and DBParm values.

The minimum value specified in the SetTransPool function must be less than or equal to the maximum value. When the minimum value is less than the maximum and the number of transactions in the pool is greater than the minimum, PowerBuilder physically terminates connections for which a DISCONNECT statement is issued until the minimum number is reached.

The maximum value specified for a transaction pool limits the total number of database connections made by the application. When the transaction pool is full, each attempt to connect will fail after the timeout interval has been exceeded.

To set up transaction pooling for an application, you need to issue SetTransPool before establishing any connections to the database.

Examples

A distributed PowerBuilder server application that services a high volume of short database transactions to the same data source could issue the following statement in its application Open event:

```
server_app.SetTransPool(12,16,10)
```

This statement specifies that up to 16 database connections will be supported through this server, and that 12 connections will be kept open once successfully connected. When the maximum number of connections has been reached, each subsequent connection request will wait for up to 10 seconds for a connection in the pool to become available. After 10 seconds, the server will return an error to the client.

The following statement specifies that up to 8 database connections will be allowed through this server, and that all 8 will be kept open once successfully connected. When the transaction pool is full, each subsequent connection request will immediately result in an error:

```
server_app.SetTransPool(8,8,0)
```

See also

SetTrans
SetTransObject

SetValidate

Description

Sets the input validation rule for a column in a DataWindow control or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetValidate** (*column, rule*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If any argument's value is NULL, SetValidate returns NULL.

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.

FOR INFO For more information, see the *DataWindow Reference* and the *PowerBuilder User's Guide*.

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 zero:

```
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 value list or code table for a column in a DataWindow control or DataStore. (A value list is called a code table when it has both display and data values.) SetValue does not affect the data stored in the column.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**SetValue** (*column*, *index*, *value*)

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow
column	The column that contains the value list or code table. *Column* can be a column number (integer) or a column name (string). The edit style of the column can be DropDownListBox, Edit, or RadioButton. SetValue has no effect when *column* has the EditMask or dddw edit style
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. If any argument's value is NULL, SetValue returns NULL.

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 a 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 (or DataStore), which is referred to as the primary DataWindow, with another DataWindow control (or DataStore), referred to as 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, DataStore objects, and child DataWindows

Syntax

dwprimary.**ShareData** (*dwsecondary*)

Argument	Description
dwprimary	The name of the primary DataWindow. The primary DataWindow is 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; which is the control *dwprimary* will share the data with. The secondary DataWindow 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. If any argument's value is NULL, ShareData returns NULL.

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
```

WHERE clause in secondary has no effect
The WHERE clause in the DataWindow object in the secondary DataWindow control has no effect on the number of rows returned. The number of rows returned to both DataWindow controls is determined by the WHERE clause in the primary DataWindow object.

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.

To turn off sharing in a primary or secondary DataWindow, call the ShareDataOff function. When sharing is turned off for the primary DataWindow, the secondary DataWindows are disconnected and the data disappears. However, turning off sharing for a secondary DataWindow does not affect the data in the primary DataWindow or other secondary DataWindows.

When you call functions in either the primary or secondary DataWindow 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. Such functions include:

DeleteRow
Filter
GetSQLSelect
ImportFile
ImportString
ImportClipboard
InsertRow
ReselectRow
Reset
Retrieve
SetFilter
SetSort
SetSQLSelect
Sort
Update

Computed fields in secondary DataWindow controls
A secondary DataWindow control can only have data which is in the primary DataWindow control. If you add a computed field to a secondary control, it will not display when you run the application unless you also add it to the primary control.

Query mode and secondary DataWindows
When you are sharing data, you cannot turn on query mode for a secondary DataWindow. Trying to set the QueryMode or QuerySort properties results in an error.

To share data between a DataStore or DataWindow and a RichTextEdit control, use the DataSource function.

Crosstab DataWindows
You cannot use ShareData with Crosstab DataWindows.

Distributed applications
You cannot share data between a DataWindow control in a client application and a DataStore in a server application.

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 or DataStore.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**ShareDataOff** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If *dwcontrol* is NULL, ShareDataOff returns NULL.

Usage

Two or more DataWindow controls (or DataStores) can share data. 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 DataWindows are disconnected and no longer contain data.

Examples

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

SharedObjectDirectory

Description

Retrieves the list of objects that have been registered for sharing.

This function is used primarily in distributed applications.

Syntax

SharedObjectDirectory (*instancenames* {, *classnames* })

Argument	Description
instancenames	An unbounded array of type string in which you want to store the names of objects that have been registered for sharing
classnames (optional)	An unbounded array of type string in which you want to store the class names of objects registered for sharing

Return value

ErrorReturn. Returns one of the following values:

- Success! — The function succeeded
- FeatureNotSupportedError! — This function is not supported on Windows 3.*x*

Usage

Shared objects are accessible only from the server's main session and from the client sessions created for each client connection on the server. Client applications cannot access a shared object directly. Therefore, to retrieve the list of registered objects, you need to call this function inside the server's execution context or in a client session on the server.

Examples

In this example, the server application retrieves the list of shared objects and their class names:

```
integer status
string InstanceNames[]
string ClassNames[]

status = SharedObjectDirectory(InstanceNames, &
   ClassNames)
```

See also

SharedObjectRegister

SharedObjectGet

Description

Gets a reference to a shared object instance.

This function is used primarily in distributed applications.

Syntax

SharedObjectGet (*instancename* , *objectinstance*)

Argument	Description
instancename	The name of a shared object instance to which you want to obtain references. The name you specify must match the name given to the object instance when it was first registered with the SharedObjectRegister function
objectinstance	An object variable in which you want to store an instance of a shared object

Return value

ErrorReturn. Returns one of the following values:

- Success! — The function succeeded
- FeatureNotSupportedError! — This function is not supported on Windows 3.*x*
- SharedObjectCreateInstanceError! — The local reference to the shared object could not be created
- SharedObjectNotExistsError! — The instance name has not been registered

Usage

SharedObjectGet retrieves a reference to an object that was created with SharedObjectRegister. Once you've obtained the object reference, you can access the object's methods and properties just as you would with any other object.

Shared objects are accessible only from the server's main session and from the client sessions created for each client connection on the server. Client applications cannot access a shared object directly. Therefore, to access a shared object, you need to call the SharedObjectGet function inside the server's execution context or in a client session on the server.

Examples

In this example the server application gets a shared object instance for the object registered as share1 and stores this instance in the variable shared_object. The server then uses the object instance to call the refresh_custlist funtion:

```
uo_customers shared_object
s_customer customers[]
```

```
long ll_rowcount
int result

SharedObjectGet("share1", shared_object)

ll_rowcount=shared_object.refresh_custlist(customers)
```

See also

SharedObjectRegister

SharedObjectRegister

Description

Registers a user object so that it can be shared.

This function is used primarily in distributed applications.

Syntax

SharedObjectRegister (*classname* , *instancename*)

Argument	Description
classname	The name of the user object that you want to share
instancename	The name you want to assign to the shared object instance

Return value

ErrorReturn. Returns one of the following values:

- Success! — The function succeeded
- FeatureNotSupportedError! — This function is not supported on Windows 3.*x*
- SharedObjectExistsError! — The instance name has already been used
- SharedObjectCreateInstanceError! — The object could not be created
- SharedObjectCreatePBSessionError! — The shared object session could not be created

Usage

When you call the SharedObjectRegister function, PowerBuilder opens a separate runtime session for the shared object and creates the shared object. The name you specify for the object instance provides a way for you to access the object instance with the SharedObjectGet function.

Shared objects are accessible only from the server's main session and from the client sessions created for each client connection on the server. Client applications cannot access a shared object directly. Therefore, to register an object to be shared, you need to call the SharedObjectRegister function inside the server's execution context or in a client session on the server.

Examples

In this example the server application registers the user object uo_customers so that it can be shared. The name assigned to the shared object instance is share1. After registering the object, the server uses the SharedObjectGet function to store an instance of the object in an object variable:

```
SharedObjectRegister("uo_customers", "share1")
SharedObjectGet("share1",shared_object)
```

See also

SharedObjectGet
SharedObjectUnregister

SharedObjectUnregister

Description

Unregisters a user object that was previously registered.

This function is used primarily in distributed applications.

Syntax

SharedObjectUnregister (*instancename*)

Argument	Description
instancename	The name assigned to the shared object instance when it was first registered

Return value

ErrorReturn. Returns one of the following values:

- Success! — The function succeeded
- FeatureNotSupportedError! — This function is not supported on Windows 3.*x*
- SharedObjectNotExistsError! — The instance name has not been registered

Usage

This function marks a shared object for destruction. But the object is not actually destroyed until there are no more references to the object.

Shared objects are accessible only from the server's main session and from the client sessions created for each client connection on the server. Client applications cannot access a shared object directly. Therefore, to unregister a shared object, you need to call the SharedObjectUnregister function inside the server's execution context or in a client session on the server.

Examples

In this example the server application unregisters the object instance called share1:

```
SharedObjectUnregister("share1")
```

See also

SharedObjectRegister

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** ( )

Argument	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. If *objectname* is NULL, Show returns NULL.

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 property 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 menu selection 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

ShowHeadFoot

Description

Displays the panels for editing the header and footer in a RichTextEdit control or hides the panels and returns to editing the main text.

Applies to

RichTextEdit controls and DataWindow controls with the RichTextEdit presentation style

Syntax

rtename.**ShowHeadFoot** (*editheadfoot*)

Argument	Description
rtename	The name of the RichTextEdit or DataWindow control for which you want to edit header and footer information
editheadfoot	A boolean value specifying the editing panel to display. Values are: ♦ TRUE — Display the header and footer editing panels ♦ FALSE — Display the detail editing panel for the document body

Return value

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

Usage

ShowHeadFoot takes effect when the control is in preview mode or when it is in edit mode for the main text. If the control is in preview mode, calling ShowHeadFoot returns to edit mode. The value of *editheadfoot* determines whether the main text or the header and footer panels display.

The header and footer can include input fields for page numbers and dates. Scripts for the PrintHeader and PrintFooter events can provide values for these fields.

For a DataWindow control, ShowHeadFoot has no effect if the DataWindow object doesn't have the RichTextEdit presentation style.

Examples

This example displays the header and footer editing panels, allowing the user to specify their contents:

```
rte_1.ShowHeadFoot(TRUE)
```

See also

Preview

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* })

Argument	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; do not specify a value for *typeid* ♦ 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!. Do not specify *typeid* when *helpcommand* is Index!

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. ShowHelp returns -1 if you specify *typeid* when *helpcommand* is Index!. If any argument's value is NULL, ShowHelp returns NULL.

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 INFO For information on how to create Windows Help files, see your Windows documentation.

Examples

This statement displays the Help index in the INQ.HLP file:

```
ShowHelp("C:\PB\INQ.HLP", Index!)
```

On Macintosh On Macintosh, the filename in the preceding code might look like this:

```
ShowHelp("HD:PowerBuilder Apps:INQ.HLP", Index!)
```

On UNIX On UNIX, the filename in the preceding code might look like this:

```
ShowHelp("/export/home/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

Reports whether a number is negative, zero, or positive.

Syntax

Sign (*n*)

Argument	Description
n	The number for which you want to find out the sign

Return value

Integer. Returns a number (-1, 0, or 1) indicating the sign of *n*. If *n* is NULL, Sign returns NULL.

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 the *DataWindow Reference*

SignalError

Description

Causes a SystemError event at the application level.

Syntax

SignalError ({ *number* }, { *text* })

Argument	Description
number (optional)	The integer (stored in the number property of the Error object) to be used in the message object
text (optional)	The string (stored in the text property of the Error object) to be used in the message object

Return value

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

Usage

During development you can use SignalError to test error-processing scripts.

You can call PopulateError to populate the Error object and call SignalError without arguments. You can examine how the SystemError event script handles the forced error.

If you pass the optional *number* and *text* arguments to SignalError, it populates all the fields in the Error object and then triggers a SystemError event.

In an application, SignalError can also be useful. 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 the execution-time error numbers that will be assigned to the Number property of the Error object when an application error occurs, see the *PowerBuilder User's Guide*.

Examples

These statements set values in the Error object and then trigger a SystemError event so the error processing for these values can be tested:

```
int error_number
string error_text

Error.Number = 1010
Error.Text = "Salary must be a positive number."
Error.Windowmenu = "w_emp"
```

```
error_number = Error.Number
error_text = Error.Text

SignalError(error_number, error_text)
```

See also PopulateError

Sin

Description

Calculates the sine of an angle.

Syntax

Sin (*n*)

Argument	Description
n	The angle (in radians) for which you want the sine

Return value

Double. Returns the sine of *n*. If *n* is NULL, Sin returns NULL.

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 the *DataWindow Reference*

Sort

Sorts rows in a DataWindow control, DataStore, or child DataWindow, or items in a TreeView or ListView control.

To sort	Use
Rows in a DataWindow control, DataStore, or child DataWindow	Syntax 1
Items in a TreeView	Syntax 2
Items in a ListView	Syntax 3

Syntax 1 For DataWindows, DataStores, and child DataWindows

Description

Sorts the rows in a DataWindow control or DataStore using the DataWindow's current sort criteria.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**Sort** ( )

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, 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. If *dwcontrol* is NULL, Sort returns NULL.

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.

When the Retrieve As Needed option is set, the Sort function cancels its effect. Sort causes all rows to be retrieved so that they are sorted correctly.

Sort has no effect on the DataWindows in a composite report.

Sorting and groups
To sort a DataWindow object with groups, call GroupCalc after you call Sort.

FOR INFO For information on letting the user specify sort criteria using the PowerBuilder built-in dialog box, see SetSort.

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.SetRedraw(false)
dw_employee.SetSort("#1 A, #2 D")
dw_employee.Sort()
dw_employee.SetRedraw(true)
```

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.SetRedraw(false)
dw_depts.Sort()
dw_depts.GroupCalc()
dw_depts.SetRedraw(true)
```

See also

GroupCalc
SetSort

Syntax 2

For TreeView controls

Description

Sorts the children of an item in a TreeView control.

Applies to

TreeView controls

Syntax

treeviewname.**Sort** (*itemhandle* , *sorttype*)

Argument	Description
treeviewname	The name of the TreeView control in which you want to sort items

Argument	Description
itemhandle	The item for which you want to sort its children
sorttype	The sort method you want to use. Valid values are: Ascending! Descending! UserDefinedSort!

Return value

Integer. Returns 1 if it succeeds and -1 if it fails.

Usage

The Sort function only sorts the immediate level beneath the specified item. If you want to sort multiple levels, use SortAll.

If you specify UserDefinedSort! as your *sorttype*, define your sort criteria in the Sort event of the TreeView control.

The Sort function can't sort level 1 of a TreeView. However, level 1 is sorted automatically when the TreeView's SortType property calls for sorting.

Examples

This example sorts the children of the current TreeView item:

```
long ll_tvi
ll_tvi = tv_foo.FindItem(CurrentTreeItem! , 0)
tv_foo.SetRedraw(false)
tv_foo.Sort(ll_tvi , Ascending!)
tv_foo.SetRedraw(true)
```

See also

SortAll

Syntax 3 For ListView controls

Description

Sorts items in ListView controls.

Applies to

ListView controls

Syntax

listviewname.**Sort** (*sorttype*, { *column* })

Argument	Description
listviewname	The ListView in which you want to sort items

Argument	Description
sorttype	The method you want to use when you sort the ListView items. Values are: Ascending! Descending! Unsorted! UserDefinedSort!
column (optional)	The number of the column by which you wish to sort the ListView items

Return value

Integer. Returns 1 if it succeeds and -1 if it fails.

Usage

The default sort is alphanumeric.

If you do not specify a column to sort, the first column is sorted.

Examples

This example sorts the items in column three of a ListView:

```
lv_list.SetRedraw(false)
lv_list.Sort(Ascending! , 3)
lv_list.SetRedraw(true)
```

See also

SortAll

SortAll

Description

Sorts all the levels below an item in the TreeView item hierarchy.

Applies to

TreeView controls

Syntax

treeviewname.**SortAll** (*itemhandle*, *sorttype*)

Argument	Description
treeviewname	The TreeView control in which you want to sort the subsequent levels in an item's hierarchy
itemhandle	The item for which you want to sort all the levels below it
sorttype	The sort method you want to use. Values are: Ascending! Descending! Unsorted! UserDefinedSort!

Return value

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

Usage

If you specify UserDefinedSort! as your *sorttype*, define your sort criteria in the Sort event of the TreeView control.

The SortAll function can't sort level 1 of a TreeView. However, level 1 is sorted automatically when the TreeView's SortType property calls for sorting.

Examples

This example sorts the subsequent levels recursively under the current TreeView item:

```
long ll_tvi

//Find the current treeitem
ll_tvi = tv_list.FindItem(CurrentTreeItem! , 0)

//Sort all children
tv_list.SortAll(ll_tvi , Ascending!)
```

This example recursively sorts the entire TreeView control:

```
long ll_tvi

//Find the root treeitem
ll_tvi = tv_list.FindItem(RootTreeItem! , 0)
```

```
//Sort all children
tv_list.SortAll(ll_tvi , Ascending!)
```

See also Sort

Space

Description

Builds a string of the specified length whose value consists of spaces.

Syntax

Space (*n*)

Argument	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. If *n* is NULL, Space returns NULL.

Examples

This statement puts a string whose value is four 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 the *DataWindow Reference*

Sqrt

Description

Calculates the square root of a number.

Syntax

Sqrt (*n*)

Argument	Description
n	The number for which you want the square root

Return value

Double. Returns the square root of *n*. If *n* is NULL, Sqrt returns NULL.

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 the *DataWindow Reference*

Start

Start has two syntaxes.

To	Use
Execute a pipeline object	Syntax 1
Activate a timing object	Syntax 2

Syntax 1

For executing pipeline objects

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 a property of a user object inherited from the pipeline system object.

Applies to

Pipeline objects

Syntax

pipelineobject.**Start** (*sourcetrans*, *destinationtrans*, *errordatawindow* {, *arg1*, *arg2*,..., *argn* })

Argument	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 (optional)	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

If any argument's value is NULL, Start returns NULL.

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 properties 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, menus, 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 property. 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 INFO For information about specifying the pipeline object in the Data Pipeline painter and how the settings affect committing, see the *PowerBuilder User's Guide*. For more information on using a pipeline in an application, see *Application Techniques*.

When you dynamically assign the pipeline object to the user object's DataObject property, you must remember to include the pipeline object in a dynamic library when you build your application's executable.

Examples

The following script creates an instance of the pipeline user object, assigns a pipeline object to the pipeline user object's DataObject property, 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

Syntax 2

For activating timing objects

Description

Activates a timing object causing a Timer event to occur repeatedly at the specified interval.

Syntax

timingobject.**Start** (*interval*)

Argument	Description
timingobject	The name of the timing object you want to activate
interval	An expression of type double specifying the number of seconds that you want between timer events. The *interval* can be a whole number or fraction greater than 0 and less than or equal to 4,294,967 seconds (less than or equal to 65 seconds on Windows 3.1). An interval of 0 is invalid

Return value

Integer. Returns 1 if it succeeds and -1 if the timer is already running, the interval specified is invalid, or there are no system timers available.

Usage

This syntax of the Start function is used to activate a nonvisual timing object. Timing objects can be used to trigger a Timer event that is not associated with a PowerBuilder window, and they are therefore useful for distributed PowerBuilder servers or shared objects that do not have a window for each client connection.

A timing object is a standard class user object inherited from the Timing system object. Once you have created a timing object and coded its timer event, you can create any number of instances of the object within the constraints of your operating system. An operating system supports a fixed number of timers; for example, on Windows 3.1 the limit is 32. Some of those timers will already be in use by PowerBuilder and other applications and by the operating system itself.

To activate an instance of the timing object, call the Start function, specifying the *interval* that you want between Timer events. The Timer event of that instance will be triggered as soon as possible after the specified interval, and will continue to be triggered until you call the Stop function on that instance of the timing object or the object is destroyed.

When the Timer event occurs

The *interval* specified for the Start function is the minimum interval between Timer events. All other posted events will occur before the Timer event.

The resolution of the interval depends on your operating system. For example, on Windows 3.1 the finest granularity is approximately 0.055 seconds.

You can determine what the timing interval is and whether a timer is running by accessing the timing object's Interval and Running properties. These properties are read-only. You must stop and restart a timer in order to change the value of the timing interval.

Garbage collection

If a timing object is running, it will not be subject to garbage collection. Garbage collection can occur only if the timing object is not running and there are no references to it.

Examples

Example 1 Suppose you have a distributed application in which the local client performs some processing, such as calculating the value of a stock portfolio, based on values in a database. The client requests a user object on a remote server to retrieve the data values from the database.

Create a standard class user object on the server called uo_timer, inherited from the Timing system object, and code its Timer event to refresh the data. Then the following code creates an instance, MyTimer, of the timing object uo_timer. The Start function activates the timer with an interval of 60 seconds so that the request to the server is issued at 60-second intervals:

```
uo_timer MyTimer

MyTimer = CREATE uo_timer
MyTimer.Start(60)
```

Example 2 The following example uses a timing object as a shared object in a window that has buttons for starting a timer, getting a hit count, stopping the timer, and closing the window. Status is shown in a single line edit called sle_state. The timing object, uo_timing, is a standard class user object inherited from the Timing system object. It has one instance variable that holds the number of times a connection is made:

```
long il_hits
```

The timing object uo_timing has three functions:

- of_connect increments il_hits and returns an integer (this example omits the connection code for simplicity):

```
il_hits++
// connection code omitted
RETURN 1
```

- of_hitcount returns the value of il_hits:

```
RETURN il_hits
```

- of_resetcounter resets the value of the counter to 0:

```
il_hits = 0
```

The timer event in uo_timing calls the of_connect function:

```
integer li_err

li_err = This.of_connect()
IF li_err <> 1 THEN
   MessageBox("Timer Error", "Connection failed ")
END IF
```

When the main window (w_timer) opens, its Open event script registers the uo_timing user object as a shared object:

```
ErrorReturn result
```

```
string ls_result

SharedObjectRegister("uo_timing","Timing")
result = SharedObjectGet("Timing", iuo_timing)
// convert enumerated type to string
ls_result = of_converterror(result)

IF result = Success! THEN
   sle_stat.text = "Object Registered"
ELSE
   MessageBox("Failed", "SharedObjectGet failed, " &
      + "Status code: "+ls_result)
END IF
```

The Start Timer button starts the timer with an interval of five seconds:

```
double ld_interval
integer li_err

IF (isvalid(iuo_timing)) THEN
   li_err = iuo_timing.Start(5)
   ld_interval = iuo_timing.interval
   sle_2.text = "Timer started. Interval is " &
      + string(ld_interval) + " seconds"
// disable Start Timer button
      THIS.enabled = FALSE
ELSE
   sle_2.text = "No timing object"
END IF
```

The Get Hits button calls the of_hitcount function and writes the result in a single line edit:

```
long ll_hits

IF (isvalid(iuo_timing)) THEN
   ll_hits = iuo_timing.of_hitcount()
   sle_hits.text = string(ll_hits)
ELSE
   sle_hits.text = ""
   sle_stat.text = "Invalid timing object..."
END IF
```

The Stop Timer button stops the timer, reenables the Start Timer button, and resets the hit counter:

```
integer li_err

IF (isvalid(iuo_timing)) THEN
   li_err = iuo_timing.Stop()

   IF li_err = 1 THEN
      sle_stat.text = "Timer stopped"
      cb_start.enabled = TRUE
      iuo_timing.of_resetcounter()
   ELSE
      sle_stat.text = "Error - timer could " &
         not be stopped"
   END IF

ELSE
   sle_stat.text = "Error - no timing object"
END IF
```

The Close button checks that the timer has been stopped and closes the window if it has:

```
IF iuo_timing.running = TRUE THEN
   MessageBox("Error","Click the Stop Timer " &
      + "button to clean up before closing")
ELSE
   close(parent)
END IF
```

The Close event for the window unregisters the shared timing object:

```
SharedObjectUnregister("Timing")
```

The of_converterror window function converts the ErrorReturn enumerated type to a string. It takes an argument of type ErrorReturn:

```
string ls_result

CHOOSE CASE a_error
CASE Success!
   ls_result = "The function succeeded"
CASE FeatureNotSupportedError!
   ls_result = "Not supported on this platform"
CASE SharedObjectExistsError!
   ls_result = "Instance name already used"
CASE MutexCreateError!
   ls_result = "Locking mechanism unobtainable"
```

```
CASE SharedObjectCreateInstanceError!
   ls_result = "Object could not be created"
CASE SharedObjectCreatePBSessionError!
   ls_result = "Could not create context session"
CASE SharedObjectNotExistsError!
   ls_result = "Instance name not registered"
CASE ELSE
   ls_result = "Unknown Error Code"
END CHOOSE

RETURN ls_result
```

See also

Stop

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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

StartHotLink (*location, applname, topic*)

Argument	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

If any argument's value is NULL, StartHotLink returns NULL.

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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

StartServerDDE ({ *windowname*, } *applname*, *topic* {, *item* })

Argument	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. If any argument's value is NULL, StartServerDDE returns NULL.

Usage

When a DDE client application sends a DDE request, the request includes one of the items you've declared that 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.

This event is	Triggered when	You call this function in the event script	To do this
RemoteHotLinkStart	A client sends a request for a hot link		
RemoteExec	A client sends a command to your application	GetCommandDDEOrigin GetCommandDDE	To find out what client application sent the command To obtain the command
RemoteSend	the client sends data	GetDataDDEOrigin GetDataDDE	To find out what client application sent the data To obtain the data
RemoteRequest	The client application requests data from your server application	SetDataDDE RespondRemote	To send the requested data To acknowledge the request
RemoteHotLinkStop	The client wants to terminate the hot link		

Examples

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

StartServerDDE

State

Description

Determines whether an item in a listbox control is highlighted.

Applies to

ListBox and PictureListBox controls

Syntax

listboxname.**State** (*index*)

Argument	Description
listboxname	The name of the ListBox or PictureListBox 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. If any argument's value is NULL, State returns NULL.

Usage

The State and SetState functions are meant for a listbox that allows multiple selections (its MultiSelect property 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

Stop

Description

Deactivates a timing object.

Syntax

timingobject.**Stop** ( )

Argument	Description
timingobject	The name of the timing object you want to deactivate

Return value

Integer. Returns 1 if it succeeds and -1 if the timer is not running or could not be stopped.

Usage

Use this function to deactivate a timing object. A stopped timer can be reactivated with the Start function.

Examples

This statement stops the timing object instance MyTimer:

```
MyTimer.Stop()
```

See also

Start

StopHotLink

Description

Terminates a hot link with a DDE server application.

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

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Caution
All arguments must match the arguments in an earlier StartHotLink call.

Syntax

StopHotLink (*location, applname, topic*)

Argument	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

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

If any argument's value is NULL, StopHotLink returns NULL.

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

StopListening

Description

Instructs a server application to stop listening for client connections.

This function applies to distributed applications only.

Applies to

Transport objects

Syntax

transportobject.**StopListening** ( )

Argument	Description
transportobject	The name of the Transport object used to process client requests for connections

Return value

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

50 Distributed service error
52 Distributed communications error

Usage

After calling the StopListening function, the server application needs to destroy the Transport object.

Examples

In this example, the server application stops listening for client requests through the mytransport Transport object:

```
mytransport.StopListening()
destroy mytransport
```

See also

Listen

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.

On UNIX platforms, this and other DDE functions have effect only if the server and client applications are developed using PowerBuilder or compiled using Wind/U from Bristol Technology.

Syntax

StopServerDDE ({ *windowname*, } *applname*, *topic*)

Argument	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. If any argument's value is NULL, StopServerDDE returns NULL.

Caution
The arguments applname and topic must match the arguments in a prior StartServerDDE call.

Examples

This statement causes the PowerBuilder application MyPBApp to stop acting as a server:

```
StopServerDDE(w_emp, "MyPBApp", "System")
```

See also

StartServerDDE

String

String has two syntaxes.

To	Use
Format data as a string according to a specified display format mask	Syntax 1
Convert a blob to a string	Syntax 2

Syntax 1

For formatting data

Description

Formats data, such as time or date values, according to a 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* })

Argument	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. *Data* can also be an Any variable containing one of these data types
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. (For details on each data type, see Usage)

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, *format* is not a valid mask, or *data* is an incompatible data type.

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 INFO For more information on specifying display formats, see the *PowerBuilder User's Guide*. Note that, although a format can include color specifications, the colors are ignored 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.

Times and dates from a DataWindow control
When you call GetItemTime or GetItemString as an argument for the String function and don't specify a display format, the value is formatted as a DateTime value. This statement returns a string like "2/26/96 00:00:00":

```
String(dw_1.GetItemTime(1, "start_date"))
```

International deployment When you use String to format a date and the month is displayed as text (for example, the display format includes "mmm"), the month is in the language of the deployment kit (DDDK) available when the application is run. If you have installed a localized DDDK in the development environment or on a user's machine, then on that machine, the month in the resulting string will be in the language of the localized DDDK.

FOR INFO For information about the localized deployment kits, which are available in French, German, Italian, Spanish, Dutch, Danish, Norwegian, and Swedish, see the *PowerBuilder User's Guide*.

Message object You can also use String to extract a string from the Message object after calling TriggerEvent or PostEvent.

FOR INFO For more information, see the TriggerEvent or PostEvent functions.

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")
```

See also

String in the *DataWindow Reference*

Syntax 2

For blobs

Description

Converts data in a blob to a string value. If the blob's value is not text data, String attempts to interpret the data as characters.

Syntax

String (*blob*)

Argument	Description
blob	The blob whose value you want returned as a string. *Blob* can also be an Any variable containing a blob

Return value

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. If *blob* is NULL, String returns NULL.

Usage

You can also use String to extract a string from the Message object after calling TriggerEvent or PostEvent.

FOR INFO For more information, see the TriggerEvent or PostEvent functions.

Examples

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 the *DataWindow Reference*

SyntaxFromSQL

Description

Generates DataWindow source code based on a SQL SELECT statement.

Applies to

Transaction objects

Syntax

transaction.**SyntaxFromSQL** (*sqlselect, presentation, err*)

Argument	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 box in the DataWindow painter. Keywords are: (Default) Tabular Grid Form (for freeform) Graph Group Label Nup The Usage section lists 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). If any argument's value is NULL, SyntaxFromSQL returns NULL.

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 properties 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 property=value* ...)
DataWindow (*property=value* ...)
Column (*property=value* ...)
Group *groupby_colnum1 Fby_colnum2* ... *property* ...)
Text *property=value* ...)
Title ('*titlestring*')"

The checklists in the DataWindow object properties chapter in the DataWindow Reference identify the properties that you can use for each object keyword.

If a database column has extended attributes with font information, then font information you specify in the SyntaxFromSQL presentation string is ignored.

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
Information on DataWindow object properties in *DataWindow Reference*

SystemRoutine

Description

Provides the routine node representing the system root in a performance analysis model.

Applies to

Profiling object

Syntax

instancename.**SystemRoutine** (*theroutine*)

Argument	Description
instancename	Instance name of the Profiling object
theroutine	A value of type ProfileRoutine containing the routine node representing the system root. This argument is passed by reference

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- ModelNotExistsError!—The function failed because no model exists

Usage

Use this function to extract the routine node representing the system root in a performance analysis model. You must have previously created the performance analysis model from a trace file using the BuildModel function. The routine node is defined as a ProfileRoutine object and provides the time spent in the routine, any called routines, the number of times each routine was called, and the class to which the routine belongs.

Examples

This example provides the routine that represents the system root in a performance analysis model:

```
Profiling lpro_model
ProfileRoutine lprort_routine

lpro_model.BuildModel()
lpro_model.SystemRoutine(lprort_routine)
...
```

See also

BuildModel

TabPostEvent

Description

Posts the specified event for each tab page in a Tab control, adding them to the end of the event queues for the tab page user objects.

Applies to

Tab controls

Syntax

tabcontrolname.**TabPostEvent** (*event* {, *word*, *long* })

Argument	Description
tabcontrolname	The name of the Tab control for which you want to post events for its tab page user objects
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 a tab page user object in *tabcontrolname* and a script must exist for the event in *tabcontrolname*
word (optional)	A long value to be stored in the WordParm property 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 property of the system's Message object. When you specify a string, a pointer to the string is stored in the LongParm property, which you can access with the String function (see Usage for PostEvent)

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs, if the event is not a valid event for the tab page user object, or if a script does not exist for the event.

Examples

Suppose tab_address contains several tab pages inherited from uo_list and uo_list has a user event called ue_display. This statement posts the event ue_display for each the tab pages in tab_address:

```
tab_address.TabPostEvent("ue_display")
```

See also

TabTriggerEvent

TabTriggerEvent

Description

Triggers the specified event for each tab page in a Tab control, which executes the scripts immediately in the index order of the tab pages.

Applies to

Tab controls

Syntax

tabcontrolname.**TabTriggerEvent** (*event* {, *word*, *long* })

Argument	Description
tabcontrolname	The name of the Tab control for which you want to trigger events for its tab page user objects
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 a tab page user object in *tabcontrolname* and a script must exist for the event in *tabcontrolname*
word (optional)	A long value to be stored in the WordParm property 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 property of the system's Message object. When you specify a string, a pointer to the string is stored in the LongParm property, which you can access with the String function (see Usage for TriggerEvent)

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs, if the event is not a valid event for the tab page user object, or if a script does not exist for the event.

Examples

Suppose tab_address contains several tab pages inherited from uo_list and uo_list has a user event called ue_display. This statement executes immediately the script for ue_display for each the tab pages in tab_address:

```
tab_address.TabTriggerEvent("ue_display")
```

See also

TabPostEvent

Tan

Description

Calculates the tangent of an angle.

Syntax

Tan (*n*)

Argument	Description
n	The angle (in radians) for which you want the tangent

Return value

Double. Returns the tangent of *n*. An execution error occurs if *n* is not valid. If *n* is NULL, Tan returns NULL.

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 the *DataWindow Reference*

Text

Description

Obtains the text of an item in a listbox.

Applies to

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

listboxname.**Text** (*index*)

Argument	Description
listboxname	The name of the listbox control 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 (""). If any argument's value is NULL, Text returns NULL.

Examples

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, MultiLineEdit, and RichTextEdit controls

Syntax

editname.**TextLine** ( )

Argument	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, or RichTextEdit 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 (""). If *editname* is NULL, TextLine returns NULL.

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
SelectTextLine

Time

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	Use
Extract the time from DateTime data, or to extract a time stored in a blob	Syntax 1
Convert a string to a time	Syntax 2
Combine numbers for hours, minutes, and seconds into a time value	Syntax 3

Syntax 1 For DateTime and blob values

Description

Extracts a time value from a DateTime value or a blob.

Syntax

Time (*datetime*)

Argument	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. *Datetime* can also be an Any variable containing a DateTime or blob

Return value

Time. Returns the time in *datetime* as a time. If *datetime* does not contain a valid time or is an incompatible data type, Time returns 00:00:00.000000. If *datetime* is NULL, Time returns NULL.

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))
```

See also

Time in the *DataWindow Reference*

Syntax 2

For strings

Description

Converts a string containing a valid time into a time value.

Syntax

Time (*string*)

Argument	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. *String* can also be an Any variable containing a string or blob

Return value

Time. Returns the time in *string* as a time. If string does not contain a valid time or is an incompatible data type, Time returns 00:00:00.000000. If *string* 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 999999).

Examples

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)
```

See also

Time in the *DataWindow Reference*

Syntax 3 For integers

Description

Combines integers representing hours, minutes, seconds, and microseconds into a time value.

Syntax

Time (*hour*, *minute*, *second* {, *microsecond* })

Argument	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 32767) of the time (note that the range of values supported for this argument is less than the total range of values possible for a microsecond)

Return value

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.

Examples

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 the *DataWindow Reference*

Timer

Description

Causes a Timer event in a window to occur repeatedly 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 (*interval* {, *windowname* })

Argument	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. If any argument's value is NULL, Timer returns NULL.

Usage

Do not call the Timer function in the Timer event. The timer gets reset automatically and the Timer event will retrigger at the interval that has already been established. Call the Timer function in another event's script when you want to stop the timer or change the interval.

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 two 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

ToAnsi

Description

Converts Unicode characters to ANSI.

Platform information
The ToAnsi function is available only in the ANSI and Unicode versions of PowerBuilder.

Syntax

ToAnsi (*string*)

Argument	Description
string	A Unicode character string whose type you want to convert to an ANSI blob

Return value

Blob. Returns an ANSI blob if it succeeds and an empty blob if it fails.

Usage

You use the ToAnsi function in the ANSI version of PowerBuilder to convert a Unicode file to ANSI. You use the ToAnsi function in the Unicode version of PowerBuilder to convert a file to an ANSI blob so it can be read by an ANSI application.

Examples

This example illustrates the use of ToAnsi in the ANSI version of PowerBuilder to convert the Unicode file mydoc:

```
integer nbr
blob myfile

myfile = ToAnsi("c:\mydoc.doc")
nbr = FileOpen(string(myfile))
```

This example illustrates the use of ToAnsi in the Unicode version of PowerBuilder to convert the file mydoc as an ANSI blob:

```
blob myfile
integer nbr

nbr = FileOpen("c:\mydoc.doc", StreamMode!)
myfile = ToAnsi("c:\mydoc.doc")
FileWrite(nbr, string(myfile))
FileClose(nbr)
```

See also

ToUnicode

Today

Description

Obtains the system date and, in some cases, the system time.

Syntax

Today ( )

Return value

Date. Returns the current system date.

Usage

Although the data type of the Today function is date, it can also return the current time. This occurs when Today is used as an argument for another function and that argument allows different data types.

For example, if you call Today as an argument to the String function, String returns both the date and time when you use a date-plus-time display format. A second example: if you call Today as an argument for the SetItem function and the data type of the target column is DateTime, both the date and time are assigned to the DataWindow.

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")
```

This statement displays the current date and time in the StaticText st_date:

```
st_date.Text = String(Today(), "m/d/yy hh:mm")
```

See also

Now
Today in the *DataWindow Reference*

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 and PictureListBox controls

Syntax

listboxname.**Top** ( )

Argument	Description
listboxname	The name of the ListBox or PictureListBox 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. If *listboxname* is NULL, Top returns NULL.

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

TotalColumns

Description

Finds the number of columns in a ListView control.

Applies to

ListView controls

Syntax

listviewname.**TotalColumns** ( )

Argument	Description
listviewname	The name of the ListView control for which you want to find the number of columns

Return value

Integer. Returns the number of columns if it succeeds and -1 if an error occurs.

Usage

Use when the ListView control set to report view.

Examples

This example displays the number of columns in a List View report view in a SingleLineEdit:

```
int li_cols
li_cols = lv_list.TotalColumns()
sle_info.text = "Total columns = " + string(li_cols)
```

See also

TotalItems
TotalSelected

TotalItems

Description

Determines the total number of items in a list control.

Applies to

ListBox, DropDownListBox, PictureListBox, DropDownPictureListBox, and ListView controls

Syntax

listcontrolname.**TotalItems** ( )

Argument	Description
listcontrolname	The name of the ListBox, DropDownListBox, PictureListBox, DropDownPictureListBox, or ListView in which you want the total number of items

Return value

Integer. Returns the total number of items in *listcontrolname*. If *listcontrolname* contains no items, TotalItems returns 0. If an error occurs, it returns -1. If *listcontrolname* is NULL, TotalItems returns NULL.

Examples

If lb_Actions contains a total of five 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 list control that are selected.

Applies to

ListBox, PictureListBox, and ListView controls

Syntax

listcontrolname.**TotalSelected ( )**

Argument	Description
listcontrolname	The name of the ListBox, PictureListBox, or ListView in which you want the number of items that are selected

Return value

Integer. Returns the number of items in *listcontrolname* that are selected. If no items in *listcontrolname* are selected, TotalSelected returns 0. If an error occurs, it returns -1. If *listcontrolname* is NULL, TotalSelected returns NULL.

Usage

TotalSelected works only if the MultiSelect property of *listcontrolname* 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

ToUnicode

Description

Converts ANSI characters to Unicode.

Platform information
The ToUnicode function is available only in the Unicode version of PowerBuilder.

Syntax

ToUnicode (*blob*)

Argument	Description
blob	An ANSI blob you want to convert to a Unicode character string

Return value

String. Returns a Unicode string if it succeeds and an empty string if it fails.

Usage

You use the ToUnicode function to convert an ANSI file to Unicode so you can open it in the Unicode version of PowerBuilder.

Examples

This example illustrates the use of ToUnicode to convert the ANSI file mydoc to Unicode:

```
string myfile
integer nbr

nbr = FileOpen("c:\mydoc.doc", StreamMode!)
myfile = ToUnicode("c:\mydoc.doc")
FileWrite(nbr, string(myfile))
FileClose(nbr)
```

See also

ToAnsi

TraceBegin

Description

Inserts an activity type value in the trace file indicating that logging has begun and then starts logging all the enabled application trace activities. Before calling TraceBegin, you must have opened the trace file using the TraceOpen function.

Syntax

TraceBegin (*identifier*)

Argument	Description
identifier	A read-only string, logged to the trace file, used to identify a tracing block. If *identifier* is NULL, an empty string is placed in the trace file

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileNotOpenError!—TraceOpen has not been called yet
- TraceStartedError!—TraceBegin has already been called

Usage

The TraceBegin call inserts an activity type value of ActBegin! in the trace file to indicate that logging has begun and then begins logging all the application activities you have selected for tracing.

TraceBegin can only be called following a TraceOpen call. And all activities to be logged must be enabled using the TraceEnableActivity function before calling TraceBegin.

If you want to generate a trace file for an entire application run, you typically include the TraceBegin function in your application's open script. If you want to generate a trace file for only a portion of the application run, you typically include the TraceBegin function in the script that initiates the functionality on which you're trying to collect data.

You can use the *identifier* argument to identify the tracing blocks within a trace file. A tracing block represents the data logged between calls to TraceBegin and TraceEnd. There may be multiple tracing blocks within a single trace file if you are tracing more than one portion of the application run.

Examples

This example opens a trace file with the name you entered in a single line edit box and a timer kind selected from a dropdown listbox. It then begins logging the enabled activities for the first block of code to be traced:

```
TimerKind ltk_kind
```

```
CHOOSE CASE ddlb_timestamp.text
CASE "None"
    ltk_kind = TimerNone!
CASE "Clock"
    ltk_kind = Clock!
CASE "Process"
    ltk_kind = Process!
CASE "Thread"
    ltk_kind = Thread!
END CHOOSE

TraceOpen(sle_filename.text,ltk_kind)

TraceEnableActivity(ActESQL!)
TraceEnableActivity(ActGarbageCollect!)
TraceEnableActivity(ActObjectCreate!)
TraceEnableActivity(ActObjectDestroy!)

TraceBegin("Trace_block_1")
```

See also

TraceOpen
TraceEnableActivity
TraceEnd

TraceClose

Description

Closes the trace file.

Syntax

TraceClose ( )

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileNotOpenError!—TraceOpen has not been called yet
- FileCloseError!—The log file is full

Usage

TraceClose closes the trace file. If you have not already called TraceEnd, TraceClose will call that function before proceeding with its processing.

You typically include the TraceClose function in your application's Close script.

Examples

This example stops logging of application trace activities and then closes the open trace file:

```
TraceEnd()
TraceClose()
```

See also

TraceBegin
TraceEnd
TraceOpen

TraceDisableActivity

Description

Disables logging of the specified trace activity.

Syntax

TraceDisableActivity (*activity*)

Argument	Description
activity	A value of the enumerated data type TraceActivity that identifies the activity for which logging should be disabled. Values are: ♦ ActError!—Occurrences of system errors and warnings ♦ ActESQL!—Embedded SQL statement entry and exit ♦ ActGarbageCollect!—Start and finish of garbage collection ♦ ActLine!—Routine line hits ♦ ActObjectCreate!—Object creation entry and exit ♦ ActObjectDestroy!—Object destruction entry and exit ♦ ActProfile!—Abbreviation for the ActRoutine!, ActESQL!, ActObjectCreate!, ActObjectDestroy!, and ActGarbageCollect! values ♦ ActRoutine!—Routine entry and exit (if this value is disabled, ActLine! is automatically disabled) ♦ ActTrace!—Abbreviation for all activities except ActLine! ♦ ActUser!—Occurrences of an activity you selected

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileNotOpenError!—TraceOpen has not been called yet
- TraceStartedError!—You have called TraceDisableActivity after TraceBegin and before TraceEnd

Usage

Use this function to disable the logging of the specified trace activities. You typically use this function if you are tracing only portions of an application run (and thus you are calling TraceBegin multiple times) and you want to log different activities during each portion of the application.

Unless specifically disabled with TraceDisableActivity, activities that were previously enabled with a call to the TraceEnableActivity function remain enabled throughout the entire application run.

You must always call the TraceEnd function before calling TraceDisableActivity.

Examples

This example logs the enabled activities for the first block of code to be traced. Then it stops logging and disables two activity types for a second trace block. When logging is resumed for another portion of the application run, the activities that are not specifically disabled remain enabled until TraceClose is called:

```
TraceEnableActivity(ActESQL!)
TraceEnableActivity(ActGarbageCollect)
TraceEnableActivity(ActObjectCreate!)
TraceEnableActivity(ActObjectDestroy!)

TraceBegin("Trace_block_1")

TraceEnd()

TraceDisableActivity(ActESQL!)
TraceDisableActivity(ActGarbageCollect!)

TraceBegin("Trace_block_2")
```

See also

TraceEnd
TraceEnableActivity

TraceEnableActivity

Description

Enables logging of the specified trace activity.

Syntax

TraceEnableActivity (*activity*)

Argument	Description
activity	A value of the enumerated data type TraceActivity that identifies the activity to be logged. Values are: ♦ ActError!—Occurrences of system errors and warnings ♦ ActESQL!—Embedded SQL statement entry and exit ♦ ActGarbageCollect!—Start and finish of garbage collection ♦ ActLine!—Routine line hits (if this value is enabled, ActRoutine! is automatically enabled) ♦ ActObjectCreate!—Object creation entry and exit ♦ ActObjectDestroy!—Object destruction entry and exit ♦ ActProfile!—Abbreviation for the ActRoutine!, ActESQL!, ActObjectCreate!, ActObjectDestroy, and ActGarbageCollect! values ♦ ActRoutine!—Routine entry and exit ♦ ActTrace!—Abbreviation for all activities except ActLine!

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileNotOpenError!—TraceOpen has not been called yet
- TraceStartedError!—You have called TraceEnableActivity after TraceBegin and before TraceEnd

Usage

Call the TraceEnableActivity function following the TraceOpen function. TraceEnableActivity allows you to specify the types of activities you want logged in the trace file. The default activity type logged is a user-defined activity type identified by the value ActUser!. This activity is enabled by the TraceOpen call. You must call TraceEnableActivity to specify the activities to be logged before you call TraceBegin.

Each call to TraceOpen resets the activity types to be logged to the default (that is, only ActUser! activities are logged).

Since the ActError! and ActUser! values require the passing of strings to the trace file, you must call the TraceError and TraceUser functions to log this information.

Unless specifically disabled with a call to the TraceDisableActivity function, activities that are enabled with TraceEnableActivity remain enabled throughout the entire application run.

Examples

This example opens a trace file with the name you entered in a single line edit box and a timer kind selected from a dropdown listbox. Then it begins logging the enabled activities for the first block of code to be traced:

```
TimerKindltk_kind

CHOOSE CASE ddlb_timestamp.text
CASE "None"
   ltk_kind = TimerNone!
CASE "Clock"
   ltk_kind = Clock!
CASE "Process"
   ltk_kind = Process!
CASE "Thread"
   ltk_kind = Thread!
END CHOOSE

TraceOpen(sle_filename.text,ltk_kind)

TraceEnableActivity(ActRoutine!)
TraceEnableActivity(ActESQL!)
TraceEnableActivity(ActGarbageCollect!)
TraceEnableActivity(ActError!)
TraceEnableActivity(ActCreateObject!)
TraceEnableActivity(ActDestroyObject!)

TraceBegin("Trace_block_1")
```

See also

TraceOpen
TraceBegin
TraceDisableActivity

TraceEnd

Description

Inserts an activity type value in the trace file indicating that logging has ended and then stops logging application trace activities.

Syntax

TraceEnd ( )

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileNotOpenError!—TraceOpen has not been called yet
- TraceNotStartedError!—TraceBegin has not been called yet

Usage

The TraceEnd call inserts an activity type value of ActBegin! in the trace file to indicate that logging has ended and then stops logging all application activities that you selected for tracing.

If you have not already called TraceEnd when you call TraceClose, TraceClose will call TraceEnd before proceeding.

If you want to generate a trace file for an entire application run, you would typically include the TraceEnd function in your application's Close script. If you want to generate a trace file for only a portion of the application run, you typically include the TraceEnd function in the script that terminates the functionality on which you're trying to collect data.

Examples

This example stops logging of application trace activities and then closes the open trace file:

```
TraceEnd()
TraceClose()
```

See also

TraceOpen
TraceBegin
TraceClose
TraceDisableActivity

TraceError

Description

Logs your own error message and its severity level to the trace file if tracing of this activity type has been enabled.

Syntax

TraceError (*severity, message*)

Argument	Description
severity	A long whose value is a number you want to indicate the severity of the error
message	A string whose value is the error message you want to add to the trace file

Return value

ErrorReturn. This function always returns Success!.

If *severity* or *message* is NULL, TraceError returns NULL and no entry is made in the trace file.

Usage

TraceError logs an activity type value of ActError! to the trace file if you enabled the tracing of this type with the TraceEnableActivity function and then called the TraceBegin function. You use the TraceError function to record your own error message. It works just like the TraceUser function except that you use it to identify more severe problems. The *severity* and *message* values are passed without modification to the trace file.

Examples

This example logs an error message to the trace file when a database retrieval fails:

```
dw_1.SetTransObject(SQLCA)

TraceUser(100, "Starting database retrieval")
IF dw_1.Retrieve() = -1 THEN
   TraceError(999, "Retrieve for dw_1 failed")
ELSE
TraceUser(200, "Database retrieval complete")
END IF
```

See also

TraceEnableActivity
TraceUser

TraceOpen

Description

Opens a trace file with the specified name and enables logging of application trace activities.

Syntax

TraceOpen (*filename*, *timer*)

Argument	Description
filename	A read-only string used to identify the trace file
timer	A value of the enumerated data type TimerKind that identifies the timer. Values are: ♦ Clock!—Use the wall clock timer ♦ Process!—Use the process timer ♦ Thread!—Use the thread timer ♦ TimeNone!—Do not log timer values

Return value

ErrorReturn. Returns one of the following values:

- Success!—The function succeeded
- FileAlreadyOpenError!—TraceOpen has been called again without an intervening TraceClose
- FileOpenError!—The file could not be opened for writing
- EnterpriseOnlyFeature!—This function is only supported in the Enterprise edition of PowerBuilder.

If *filename* is NULL, TraceOpen returns NULL.

Usage

TraceOpen opens the specified trace file and enables logging of application trace activities. When it opens the trace file, TraceOpen logs the current application and library list to the trace file. It also enables logging of the default activity type, a user-defined activity type identified by the value ActUser!.

After calling TraceOpen, you can select any additional activities to be logged in the trace file using the TraceEnableActivity function. Once you have called TraceOpen and TraceEnableActivity, you must then call TraceBegin for logging to begin.

To stop logging of application trace activity, you must call the TraceEnd function followed by TraceClose to close the trace file. Each call to TraceOpen resets the logging of activity types to the default ActUser!

You typically include the TraceOpen function in your application's Open script.

Caution

If the trace file runs out of disk space, no error is generated. But logging is stopped, and the trace file cannot be used for analysis.

Timer values on Windows NT and Windows 95 By default, the time at which each activity begins and ends is recorded using the clock timer, which measures an absolute time with reference to an external activity, such as the machine's startup time. The clock timer measures time in microseconds. Depending on the speed of your machine's central processing unit, the clock timer can offer a resolution of less than one microsecond. A timer's resolution is the smallest unit of time the timer can measure.

You can also use process or thread timers, which measure time in microseconds with reference to when the process or thread being executed started. Use the thread timer for distributed applications. Both process and thread timers give you a more accurate measurement of how long the process or thread is taking to execute, but both have a lower resolution than the clock timer.

If your analysis does not require timing information, you can omit timing information from the trace file.

Timer values on Windows 3.1, UNIX, and Macintosh If you are running an executable file on Windows 3.1, UNIX, or Macintosh, specifying a different timer kind has no effect.

Collection time The timestamps in the trace file exclude the time taken to collect the trace data.

Examples

This example opens a trace file with the name you entered in a single line edit box and a timer kind selected from a dropdown listbox. Then it begins logging the enabled activities for the first block of code to be traced:

```
TimerKindltk_kind

CHOOSE CASE ddlb_timestamp.text
CASE "None"
    ltk_kind = TimerNone!
CASE "Clock"
    ltk_kind = Clock!
CASE "Process"
    ltk_kind = Process!
CASE "Thread"
```

```
        ltk_kind = Thread!
    END CHOOSE

    TraceOpen(sle_filename.text,ltk_kind)
```

See also

TraceBegin
TraceClose
TraceEnableActivity
TraceEnd

TraceUser

Description

Logs the activity type value you specify to the trace file.

Syntax

TraceUser (*info*, *message*)

Argument	Description
info	A long whose value is a reference number you want to associate with the logged activity
message	A string whose value is the activity type value you want to add to the trace file

Return value

ErrorReturn. This function always returns Success!.

If *info* or *message* is NULL, TraceUser returns NULL and no entry is made in the log file.

Usage

TraceUser logs an activity type value of ActUser! to the trace file. This is the default activity type and is enabled when the TraceOpen function is called. You use the TraceUser function to record your own message identifying a specific occurrence during an application run. For example, you may want to log the occurrences of a specific return value or the beginning and end of a body of code. TraceUser works just like the TraceError function except that you use TraceError to identify more severe problems. The *info* and *message* values are passed without modification to the trace file.

Examples

This example logs user messages to the trace file identifying when a database retrieval is started and when it is completed:

```
dw_1.SetTransObject(SQLCA)

TraceUser(100, "Starting database retrieval")
IF dw_1.Retrieve() = -1 THEN
   TraceError(999, "Retrieve for dw_1 failed")
ELSE
TraceUser(200, "Database retrieval complete")
END IF
```

See also

TraceEnableActivity
TraceError

TriggerEvent

Description

Triggers an event associated with the specified object, which executes the script for that event immediately.

Applies to

Any object

Syntax

objectname.**TriggerEvent** (*event* {, *word*, *long* })

Argument	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 property 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 property of the system's Message object. When you specify a string, a pointer to the string is stored in the LongParm property, 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*. If any argument's value is NULL, TriggerEvent returns NULL.

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.

Check return code
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")
```

Caution
Do not use this syntax unless you are certain the *long* argument contains a valid string value.

FOR INFO 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 menu selection m_File on the menu m_Appl:

```
m_Appl.m_File.TriggerEvent(Clicked!)
```

See also

Post
PostEvent
Send

TriggerPBEvent

Description

Triggers the specified user event in the child window contained in a PowerBuilder window ActiveX control.

Applies to

Window ActiveX controls

Syntax

activexcontrol.**TriggerPBEvent** (*name* {, *numarguments* {, *arguments* } })

Argument	Description
activexcontrol	Identifier for the instance of the PowerBuilder window ActiveX control. When used in HTML, this is the NAME attribute of the object element. When used in other environments, this references the control that contains the PowerBuilder window ActiveX
name	String specifying the name of the user event. This argument is passed by reference
numarguments (optional)	Integer specifying the number of elements in the *arguments* array. The default is zero
arguments (optional)	Variant array containing event arguments. In PowerBuilder, Variant maps to the Any data type. This argument is passed by reference If you specify this argument, you must also specify *numarguments*. If you do not specify this argument and the function contains arguments, populate the argument list by calling the SetArgElement function once for each argument JavaScript cannot use this argument

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage

Call this function to trigger a user event in the child window contained in a PowerBuilder window ActiveX control.

To check the PowerBuilder function's return value, call the GetLastReturn function.

JavaScript cannot use the *arguments* argument.

Examples

This JavaScript example calls the TriggerPBEvent function:

```
function triggerEvent(f) {
   var retcd;
   var rc;
   var numargs;
```

```
    var theEvent;
    var theArg;
    retcd = 0;
    numargs = 1;
    theArg = f.textToPB.value;
    PBRX1.SetArgElement(1, theArg);
    theEvent = "ue_args";
    retcd = PBRX1.TriggerPBEvent(theEvent, numargs);
    rc = parseInt(PBRX1.GetLastReturn());
    if (rc != 1) {
       alert("Error. Empty string.");
    }
    PBRX1.ResetArgElements();
}
```

This VBScript example calls the TriggerPBEvent function:

```
Sub TrigEvent_OnClick()
    Dim retcd
    Dim myForm
    Dim args(1)
    Dim rc
    Dim numargs
    Dim theEvent
    retcd = 0
    numargs = 1
    rc = 0
    theEvent = "ue_args"
    Set myForm = Document.buttonForm
    args(0) = buttonForm.textToPB.value
    retcd = PBRX1.TriggerPBEvent(theEvent, &
       numargs, args)
    rc = PBRX1.GetLastReturn()
    if rc <> 1 then
    msgbox "Error. Empty string."
    end if
end sub
```

See also

GetLastReturn
SetArgElement
InvokePBFunction

Trim

Description

Removes leading and trailing spaces from a string.

Syntax

Trim (*string*)

Argument	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. If *string* is NULL, Trim returns NULL.

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 the *DataWindow Reference*

Truncate

Description

Truncates a number to the specified number of decimal places.

Syntax

Truncate (*x*, *n*)

Argument	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 NULL if it fails or if any argument is NULL.

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 the *DataWindow Reference*

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 ( )**

Argument	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*. If *objectname* is NULL, TypeOf returns NULL.

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 five controls in the w_dept window's Control array property. 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 property:

```
object winobjecttype[]
long ll_count

FOR ll_count = 1 to UpperBound(Control[])
   winobjecttype[ll_count] = &
      TypeOf(Control[ll_count])
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 property to FALSE.

Applies to

Menu objects

Syntax

menuname.**Uncheck** ( )

Argument	Description
menuname	The fully qualified name of the menu selection from which you want to remove the checkmark, if any. The menu 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. If *menuname* is NULL, Uncheck returns NULL.

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 property instead of calling Uncheck:

menuname.Checked = FALSE

This statement:

```
m_appl.m_view.m_grid.Checked = FALSE
```

is equivalent to:

```
m_appl.m_view.m_grid.Uncheck()
```

Examples

This statement removes the checkmark next to the m_grid menu selection in the dropdown menu m_view on the menu bar m_appl:

```
m_appl.m_view.m_grid.Uncheck()
```

This example checks whether the m_grid menu selection in the dropdown menu 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, RichTextEdit, and SingleLineEdit controls

Syntax

editname.**Undo** ( )

Argument	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, RichTextEdit, 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. If *editname* is NULL, Undo returns NULL.

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*)

Argument	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 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. If any argument's value is NULL, UnitsToPixels returns NULL.

Examples

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 or DataStore. Update can also call AcceptText for the current row and column before it updates the database.

Applies to

DataWindow controls, DataStore objects, and child DataWindows

Syntax

dwcontrol.**Update** ({ *accept* {, *resetflag* } })

Argument	Description
dwcontrol	The name of the DataWindow control, DataStore, or child DataWindow that contains the information you want to use to update the database
accept (optional)	A boolean value specifying whether the DataWindow control or DataStore should automatically perform an AcceptText prior to performing the update: ♦ TRUE — (Default) Perform AcceptText. The update is canceled if the data fails validation ♦ FALSE — Do not perform AcceptText
resetflag (optional)	A boolean value specifying whether *dwcontrol* 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. If any argument's value is NULL, Update returns NULL.

Usage

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.

Test success/failure code

It is good practice to test the success/failure code after calling Update. In addition to checking the return value of Update, check the SQLNRows property of the transaction object, which indicates the number of rows affected, to make sure the update changed at least one row. Since the database vendor supplies this number, its meaning may not be the same in every DBMS.

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 transaction object for the DataWindow control or DataStore 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 or DataStore, you can use Modify to change the Update property 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 or DataStore, 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. For an example of this technique, see Modify.

If you are updating multiple DataWindow controls or DataStores 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 AND SQLCA.SQLNRows > 0 THEN
   COMMIT USING SQLCA;
ELSE
   ROLLBACK USING SQLCA;
END IF
```

See also

AcceptText
Modify
ResetUpdate
Print
SaveAs
SetTrans
SetTransObject

UpdateLinksDialog

Description

Attempts to find a file linked to an OLE container. If the linked file is not found, a dialog box tells the user and lets them bring up a second dialog box for find the file or changing the link.

Applies to

OLE controls and OLE DWObjects (objects within a DataWindow object that is within a DataWindow control)

Syntax

objectref.**UpdateLinksDialog** ( )

Argument	Description
objectref	The name of the OLE control or the fully qualified name of a OLE DWObject within a DataWindow control that contains the object for which you want to establish a link. The fully qualified name for a DWObject has this syntax: *dwcontrol*.Object.*dwobjectname*

Return value

Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage

If a container's LinkUpdateOptions property is set for automatic update, PowerBuilder tries to update the link when the OLE container is created and the object is loaded (for example, when the window is opened). If the linked file is not found, a message informs the user and he or she can choose to edit the link (for example, break the link or browse for the correct file).

UpdateLinksDialog and LinkTo are useful when a linked file has been moved and the container's LinkUpdateOptions property is set for manual update.

UpdateLinksDialog Calling this function triggers the same process that occurs for automatic update. PowerBuilder will try to find the file and if it fails it will give the user the opportunity to edit the link.

LinkTo If you want to establish a link without involving the user, call the LinkTo function. Its arguments specify the file and item you want to link. If you want to display your own dialog for selecting the linked file, you can take the information the user specifies and call the LinkTo function.

If the OLE container holds an embedded object, calling UpdateLinksDialog has no effect. It returns zero because no link is broken.

FOR INFO For more information about updating links, see *Application Techniques*.

Examples

This example looks for the linked file for an OLE control ole_report. If the file is missing, it prompts the user to display the Links dialog and edit the link:

```
ole_report.UpdateLinksDialog()
```

This example looks for the linked file for an OLE DWObject ole_word in the DataWindow control dw_customer_data. If the file is missing, the user can choose to edit the link using the Links dialog:

```
dw_customer_data.Object.ole_word.UpdateLinksDialog()
```

See also

InsertObject
LinkTo

Upper

Description

Converts all the characters in a string to uppercase.

Syntax

Upper (*string*)

Argument	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. If *string* is NULL, Upper returns NULL.

Examples

This statement returns BABE RUTH:

```
Upper("Babe Ruth")
```

See also

Lower
Upper in the *DataWindow Reference*

UpperBound

Description

Obtains the upper bound of a dimension of an array.

Syntax

UpperBound (*array* {, *n* })

Argument	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

Long. Returns the upper bound of dimension *n* of *array*. If *n* is greater than the number of dimensions of the array, UpperBound returns -1. If any argument's value is NULL, UpperBound returns NULL.

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 — A property of a window whose value identifies the menu associated with the window.
- Item — A property 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:

```
long 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** ( )

Argument	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. If *windowname* is NULL, WorkSpaceHeight returns NULL.

Usage

The workspace height does not include the thickness of the frame, the title bar, menu bar, horizontal scrollbar, or any toolbars at the top or bottom. The workspace height includes the MicroHelp status bar.

The workspace width does not include the thickness of the frame, the vertical scrollbar, or any toolbars on the left or right.

Examples

This example returns the height of the workspace area in the w_employee window:

```
Integer Height
Height = W_employee.WorkSpaceHeight()
```

This example resizes the client area of a custom MDI frame window (that is, a frame window in which you've placed controls). P_logo is the control that's been placed on the window. The code belongs in the script for the frame's Resize event:

```
integer lw, lh
// Get the current workspace measurements
lw = This.WorkSpaceWidth()
lh = This.WorkSpaceHeight()

// Subtract the logo, MicroHelp from the height
lh = lh - (p_logo.Y + p_logo.Height)
lh = lh - MDI_1.MicroHelpHeight

// Add the distance between the top of the frame
// (just below the menu bar or toolbar, if any)
```

```
// and top of the workspace.
lh = lh + This.WorkspaceY( )

// Move the client area below the picture control
MDI_1.Move(This.WorkspaceX( ), &
    p_logo.Y + p_logo.Height)

// Resize the client area using the calculated dims
mdi_1.Resize(lw, lh)
```

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** ( )

Argument	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. If *windowname* is NULL, WorkSpaceWidth returns NULL.

Usage

The workspace height does not include the thickness of the frame, the title bar, menu bar, horizontal scrollbar, or any toolbars at the top or bottom. The workspace height includes the MicroHelp status bar.

The workspace width does not include the thickness of the frame, the vertical scrollbar, or any toolbars on the left or right.

Examples

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.

For custom MDI frames, WorkSpaceX obtains the distance between the left edge of the frame window and the left side of the workspace area.

Applies to

Window objects

Syntax

windowname.**WorkSpaceX** ( )

Argument	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. If *windowname* is NULL, WorkSpaceX returns NULL.

Usage

The workspace area is the area between the sides of the window (not including the thickness of the frame or the vertical scrollbar, 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 scrollbar, if any).

Examples

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.

For custom MDI frames, WorkSpaceY obtains the distance from the top of the frame window and the top of the workspace area. The top of the frame window is the lower edge of the menu bar or toolbar, if any.

Applies to

Window objects

Syntax

windowname.**WorkSpaceY** ( )

Argument	Description
windowname	The name of the window for which you want the distance between the top of the workspace area and the top of 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. If *windowname* is NULL, WorkSpaceY returns NULL.

Usage

The workspace area is the area between the sides of the window (not including the thickness of the frame or the vertical scrollbar, 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 scrollbar, if any).

Examples

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.

Platform information
This and other OLE functions have no effect on Macintosh and UNIX.

Applies to

OLEStream objects

Syntax

olestream.**Write** (*dataforstream*)

Argument	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 the number of characters or bytes written 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

If any argument's value is NULL, Write returns NULL.

Examples

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

Length
Open
Read
Seek

Year

Description

Determines the year of a date value.

Syntax

Year (*date*)

Argument	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. If *date* is NULL, Year returns d.

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.

Examples

This statement returns 1995:

```
Year(1995-01-31)
```

See also

Day
Month
Year in the *DataWindow Reference*

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, some code is processing a long task. A second window includes a button that the user can click to interrupt the loop by setting a shared boolean variable sb_interrupt. When the user clicks the button, its Clicked script sets sb_interrupt, shown here:

```
sb_interrupt = TRUE
```

The script that is doing the processing checks the shared variable sb_interrupt and interrupts the processing if it is TRUE. The Yield function allows a break in the processing so the user has the opportunity to click the button:

```
integer n
// sb_interrupt is a shared variable.
sb_interrupt = FALSE

FOR n = 1 to 3000
   Yield()
```

```
    IF sb_interrupt THEN // var set in other script
       MessageBox("Debug","Interrupted!")
       sb_interrupt = FALSE
       EXIT
    ELSE
    ... // Some processing
    END IF
NEXT
```

In this example, this script doing some processing runs in one window while users interact with controls in a second window. Without Yield, users could click in the second window, but they would not see focus change or their actions processed until the loop completed:

```
integer n

FOR n = 1 to 3000
   Yield()
   ... // Some processing
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
```

Index

Symbols

A

B

C

D

E

F

G

H

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z