Contents

About This Book ................................................................................................................................ xxv

PART 1 POWERSCRIPT TOPICS

CHAPTER 1 Language Basics.......................................................................................... 3
   Comments................................................................. 3
   Identifier names.................................................... 5
   Labels........................................................................... 6
   Special ASCII characters .................................... 6
   NULL values........................................................... 8
   Reserved words.................................................... 10
   Pronouns................................................................. 11
      Parent pronoun.................................................. 12
      This pronoun ...................................................... 13
      Super pronoun................................................... 14
   Statement continuation ....................................... 15
   Statement separation ........................................... 16
   White space .......................................................... 16
   Conditional compilation ....................................... 17

CHAPTER 2 Datatypes................................................................................................. 21
   Standard datatypes ............................................. 21
   The Any datatype ................................................ 26
   System object datatypes .................................... 29
   Enumerated datatypes ....................................... 30
   PowerBuilder datatypes in EAServer ................. 31

CHAPTER 3 Declarations......................................................................................... 33
   Declaring variables.............................................. 33
      Where to declare variables.............................. 34
      About using variables ..................................... 35
      Syntax of a variable declaration .................... 37
Contents

Declaring constants................................................................. 47
Declaring arrays....................................................................... 48
Values for array elements ....................................................... 51
Size of variable-size arrays .................................................... 52
More about arrays .................................................................. 54
Declaring external functions.................................................... 57
Datatypes for external function arguments............................ 62
Calling external functions ....................................................... 64
Defining source for external functions ..................................... 64
Declaring DBMS stored procedures as remote procedure calls .... 65

CHAPTER 4 Operators and Expressions................................................. 69
Operators in PowerBuilder ....................................................... 69
Arithmetic operators in PowerBuilder ...................................... 70
Relational operators in PowerBuilder ...................................... 72
Concatenation operator in PowerBuilder .................................. 73
Operator precedence in PowerBuilder expressions .................... 74
Datatype of PowerBuilder expressions ..................................... 75
Numeric datatypes in PowerBuilder ......................................... 75
String and char datatypes in PowerBuilder .............................. 78

CHAPTER 5 Structures and Objects ....................................................... 79
About structures ..................................................................... 79
About objects ......................................................................... 80
About user objects .................................................................. 81
Instantiating objects ............................................................... 83
Using ancestors and descendants .......................................... 84
Garbage collection ................................................................. 84
User objects that behave like structures ................................. 85
Assignment for objects and structures ..................................... 86
Assignment for structures ....................................................... 86
Assignment for objects .......................................................... 87
Assignment for autoinstantiated user objects ......................... 88

CHAPTER 6 Calling Functions and Events ............................................... 91
About functions and events ..................................................... 91
Finding and executing functions and events ......................... 94
Finding functions ................................................................. 94
Finding events ................................................................. 95
Triggering versus posting functions and events .................... 96
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static versus dynamic calls</td>
<td>98</td>
</tr>
<tr>
<td>Static calls</td>
<td>98</td>
</tr>
<tr>
<td>Dynamic calls</td>
<td>99</td>
</tr>
<tr>
<td>Overloading, overriding, and extending functions and events</td>
<td>104</td>
</tr>
<tr>
<td>Overloading and overriding functions</td>
<td>104</td>
</tr>
<tr>
<td>Extending and overriding events</td>
<td>105</td>
</tr>
<tr>
<td>Passing arguments to functions and events</td>
<td>106</td>
</tr>
<tr>
<td>Passing objects</td>
<td>106</td>
</tr>
<tr>
<td>Passing structures</td>
<td>107</td>
</tr>
<tr>
<td>Passing arrays</td>
<td>108</td>
</tr>
<tr>
<td>Using return values</td>
<td>109</td>
</tr>
<tr>
<td>Functions</td>
<td>109</td>
</tr>
<tr>
<td>Events</td>
<td>109</td>
</tr>
<tr>
<td>Using cascaded calling and return values</td>
<td>110</td>
</tr>
<tr>
<td>Syntax for calling PowerBuilder functions and events</td>
<td>111</td>
</tr>
<tr>
<td>Calling functions and events in an object’s ancestor</td>
<td>114</td>
</tr>
</tbody>
</table>

### PART 2  STATEMENTS, EVENTS, AND FUNCTIONS

#### CHAPTER 7  PowerScript Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>120</td>
</tr>
<tr>
<td>CALL</td>
<td>123</td>
</tr>
<tr>
<td>CHOOSE CASE</td>
<td>124</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>126</td>
</tr>
<tr>
<td>CREATE</td>
<td>127</td>
</tr>
<tr>
<td>DESTROY</td>
<td>130</td>
</tr>
<tr>
<td>DO...LOOP</td>
<td>131</td>
</tr>
<tr>
<td>EXIT</td>
<td>133</td>
</tr>
<tr>
<td>FOR...NEXT</td>
<td>134</td>
</tr>
<tr>
<td>GOTO</td>
<td>136</td>
</tr>
<tr>
<td>HALT</td>
<td>137</td>
</tr>
<tr>
<td>IF...THEN</td>
<td>138</td>
</tr>
<tr>
<td>RETURN</td>
<td>138</td>
</tr>
<tr>
<td>THROW</td>
<td>140</td>
</tr>
<tr>
<td>THROWS</td>
<td>142</td>
</tr>
<tr>
<td>TRY...CATCH...FINALLY...END TRY</td>
<td>143</td>
</tr>
</tbody>
</table>

#### CHAPTER 8  SQL Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using SQL in scripts</td>
<td>146</td>
</tr>
<tr>
<td>CLOSE Cursor</td>
<td>149</td>
</tr>
<tr>
<td>CLOSE Procedure</td>
<td>150</td>
</tr>
<tr>
<td>COMMIT</td>
<td>151</td>
</tr>
</tbody>
</table>
Contents

CONNECT ............................................................................. 152
DECLARE Cursor ............................................................. 153
DECLARE Procedure ....................................................... 153
DELETE ............................................................................. 155
DELETE Where Current of Cursor ..................................... 156
DISCONNECT .................................................................... 156
EXECUTE ............................................................................. 157
FETCH .................................................................................. 158
INSERT ................................................................................. 159
OPEN Cursor ........................................................................ 160
ROLLBACK ........................................................................... 160
SELECT ................................................................................ 161
SELECTBLOB ....................................................................... 162
UPDATE ................................................................................ 164
UPDATEBLOB ....................................................................... 165
UPDATE Where Current of Cursor ..................................... 166

Using dynamic SQL ............................................................ 167
Dynamic SQL Format 1 ..................................................... 171
Dynamic SQL Format 2 ..................................................... 172
Dynamic SQL Format 3 ..................................................... 173
Dynamic SQL Format 4 ..................................................... 176

CHAPTER 9 PowerScript Events ........................................... 183
About events ........................................................................ 183
Activate ............................................................................ 186
BeginDownload .............................................................. 186
BeginDrag ........................................................................... 187
BeginLabelEdit ............................................................... 190
BeginLogScan ..................................................................... 192
BeginRightDrag ............................................................... 192
BeginSync .......................................................................... 195
BeginUpload ....................................................................... 195
Clicked ............................................................................. 196
Close ................................................................................ 202
CloseQuery ........................................................................ 204
CloseUp ............................................................................ 205
ColumnClick ....................................................................... 206
ConnectMobiLink ............................................................ 207
Constructor ...................................................................... 208
DataChange ....................................................................... 209
DateChanged ....................................................................... 209
DateSelected ....................................................................... 210
DBError ............................................................................. 210
DBNotification .................................................................... 212
<table>
<thead>
<tr>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deactivate</td>
<td>215</td>
</tr>
<tr>
<td>DeleteAllItems</td>
<td>215</td>
</tr>
<tr>
<td>DeleteItem</td>
<td>216</td>
</tr>
<tr>
<td>Destructor</td>
<td>217</td>
</tr>
<tr>
<td>DisconnectMobiLink</td>
<td>217</td>
</tr>
<tr>
<td>DisplayMessage</td>
<td>218</td>
</tr>
<tr>
<td>DoubleClicked</td>
<td>219</td>
</tr>
<tr>
<td>DragDrop</td>
<td>222</td>
</tr>
<tr>
<td>DragEnter</td>
<td>227</td>
</tr>
<tr>
<td>DragLeave</td>
<td>228</td>
</tr>
<tr>
<td>DragWithin</td>
<td>229</td>
</tr>
<tr>
<td>DropDown</td>
<td>232</td>
</tr>
<tr>
<td>EndDownload</td>
<td>232</td>
</tr>
<tr>
<td>EndLabelEdit</td>
<td>233</td>
</tr>
<tr>
<td>EndLogScan</td>
<td>234</td>
</tr>
<tr>
<td>EndSync</td>
<td>235</td>
</tr>
<tr>
<td>EndUpload</td>
<td>236</td>
</tr>
<tr>
<td>Error</td>
<td>236</td>
</tr>
<tr>
<td>ErrorMessage</td>
<td>239</td>
</tr>
<tr>
<td>ExternalException</td>
<td>240</td>
</tr>
<tr>
<td>FileExists</td>
<td>243</td>
</tr>
<tr>
<td>FileMessage</td>
<td>244</td>
</tr>
<tr>
<td>Gesture</td>
<td>244</td>
</tr>
<tr>
<td>GetFocus</td>
<td>246</td>
</tr>
<tr>
<td>Help</td>
<td>247</td>
</tr>
<tr>
<td>Hide</td>
<td>248</td>
</tr>
<tr>
<td>HotLinkAlarm</td>
<td>249</td>
</tr>
<tr>
<td>Idle</td>
<td>249</td>
</tr>
<tr>
<td>InputFieldSelected</td>
<td>250</td>
</tr>
<tr>
<td>InsertItem</td>
<td>250</td>
</tr>
<tr>
<td>ItemActivate</td>
<td>251</td>
</tr>
<tr>
<td>ItemChanged</td>
<td>252</td>
</tr>
<tr>
<td>ItemChanging</td>
<td>253</td>
</tr>
<tr>
<td>ItemCollapsed</td>
<td>254</td>
</tr>
<tr>
<td>ItemCollapsing</td>
<td>255</td>
</tr>
<tr>
<td>ItemExpanded</td>
<td>256</td>
</tr>
<tr>
<td>ItemExpanding</td>
<td>257</td>
</tr>
<tr>
<td>ItemPopulate</td>
<td>258</td>
</tr>
<tr>
<td>Key</td>
<td>258</td>
</tr>
<tr>
<td>LineDown</td>
<td>260</td>
</tr>
<tr>
<td>LineLeft</td>
<td>261</td>
</tr>
<tr>
<td>LineRight</td>
<td>262</td>
</tr>
<tr>
<td>LineUp</td>
<td>263</td>
</tr>
<tr>
<td>LoseFocus</td>
<td>264</td>
</tr>
</tbody>
</table>
SyncPreview ................................................................. 315
SystemError ................................................................. 315
SystemKey ................................................................. 316
Timer ............................................................... 317
ToolBarMoved ............................................................. 319
UploadAck ................................................................. 320
UserString ................................................................. 321
ValueChanged ............................................................... 322
ViewChange ................................................................. 322
WaitForUploadAck ........................................................ 323
WarningMessage ........................................................... 323

CHAPTER 10  PowerScript Functions ............................................................... 325
Abs ................................................................. 326
ACos ............................................................... 326
Activate ................................................................. 327
AddCategory .............................................................. 329
AddColumn .............................................................. 330
AddData ................................................................. 331
AddItem ................................................................. 333
AddLargePicture .......................................................... 338
AddPicture .............................................................. 339
AddSeries ................................................................. 340
AddSmallPicture .......................................................... 341
AddStatePicture ........................................................... 342
AddToLibraryList ......................................................... 343
Arrange ................................................................. 344
ArrangeSheets ............................................................ 345
Asc ................................................................. 346
AscA ................................................................. 346
ASin ................................................................. 348
ATan ................................................................. 349
Beep ................................................................. 349
BeginTransaction ......................................................... 350
Blob ................................................................. 351
BlobEdit ............................................................... 353
BlobMid ................................................................. 354
BuildModel ............................................................... 355
Byte ................................................................. 358
Cancel ................................................................. 359
CancelSync ............................................................... 360
CanUndo ............................................................... 360
CategoryCount ............................................................. 361
CategoryName ............................................................. 362
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ceiling</td>
<td>363</td>
</tr>
<tr>
<td>ChangeDirectory</td>
<td>363</td>
</tr>
<tr>
<td>ChangeMenu</td>
<td>364</td>
</tr>
<tr>
<td>Char</td>
<td>365</td>
</tr>
<tr>
<td>CharA</td>
<td>366</td>
</tr>
<tr>
<td>Check</td>
<td>367</td>
</tr>
<tr>
<td>ChooseColor</td>
<td>368</td>
</tr>
<tr>
<td>ClassList</td>
<td>368</td>
</tr>
<tr>
<td>ClassName</td>
<td>369</td>
</tr>
<tr>
<td>Clear</td>
<td>372</td>
</tr>
<tr>
<td>ClearBoldDates</td>
<td>373</td>
</tr>
<tr>
<td>Clipboard</td>
<td>374</td>
</tr>
<tr>
<td>Close</td>
<td>376</td>
</tr>
<tr>
<td>CloseChannel</td>
<td>380</td>
</tr>
<tr>
<td>CloseTab</td>
<td>381</td>
</tr>
<tr>
<td>CloseUserObject</td>
<td>382</td>
</tr>
<tr>
<td>CloseWithReturn</td>
<td>383</td>
</tr>
<tr>
<td>CollapseItem</td>
<td>386</td>
</tr>
<tr>
<td>CommandParm</td>
<td>387</td>
</tr>
<tr>
<td>CommitTransaction</td>
<td>388</td>
</tr>
<tr>
<td>ConnectToNewObject</td>
<td>390</td>
</tr>
<tr>
<td>ConnectToNewRemoteObject</td>
<td>392</td>
</tr>
<tr>
<td>ConnectToObject</td>
<td>393</td>
</tr>
<tr>
<td>ConnectToRemoteObject</td>
<td>396</td>
</tr>
<tr>
<td>ConnectToServer</td>
<td>398</td>
</tr>
<tr>
<td>Copy</td>
<td>401</td>
</tr>
<tr>
<td>CopyRTF</td>
<td>402</td>
</tr>
<tr>
<td>Cos</td>
<td>404</td>
</tr>
<tr>
<td>Cpu</td>
<td>405</td>
</tr>
<tr>
<td>CreateDirectory</td>
<td>405</td>
</tr>
<tr>
<td>CreateInstance</td>
<td>406</td>
</tr>
<tr>
<td>CreatePage</td>
<td>410</td>
</tr>
<tr>
<td>Cut</td>
<td>411</td>
</tr>
<tr>
<td>DataCount</td>
<td>413</td>
</tr>
<tr>
<td>DataSource</td>
<td>414</td>
</tr>
<tr>
<td>Date</td>
<td>415</td>
</tr>
<tr>
<td>DateTime</td>
<td>419</td>
</tr>
<tr>
<td>Day</td>
<td>420</td>
</tr>
<tr>
<td>DayName</td>
<td>421</td>
</tr>
<tr>
<td>DayNumber</td>
<td>422</td>
</tr>
<tr>
<td>DaysAfter</td>
<td>423</td>
</tr>
<tr>
<td>DBHandle</td>
<td>424</td>
</tr>
<tr>
<td>DebugBreak</td>
<td>425</td>
</tr>
<tr>
<td>Dec</td>
<td>425</td>
</tr>
</tbody>
</table>
Contents

DeleteCategory ............................................................... 426
DeleteColumn ............................................................... 427
DeleteColumns .............................................................. 427
DeleteData ......................................................................... 428
DeleteItem ....................................................................... 429
DeleteItems ...................................................................... 431
DeleteLargePicture .......................................................... 432
DeleteLargePictures ........................................................ 432
DeletePicture .................................................................. 433
DeletePictures ................................................................. 433
DeleteSeries .................................................................... 434
DeleteSmallPicture .......................................................... 435
DeleteSmallPictures ........................................................ 435
DeleteStatePicture ........................................................... 436
DeleteStatePictures ......................................................... 436
DestroyModel ................................................................... 437
DirectoryExists ............................................................... 437
DirList ................................................................................ 438
DirSelect .......................................................................... 440
Disable .............................................................................. 441
DisableCommit .................................................................. 442
DisconnectObject ............................................................ 443
DisconnectServer ............................................................. 444
Double ............................................................................... 445
DoVerb ............................................................................... 446
Drag .................................................................................... 447
DraggedObject ................................................................. 448
Draw ................................................................................. 450
EditLabel .......................................................................... 451
Enable ............................................................................... 452
EnableCommit ................................................................... 453
EntryList ............................................................................ 454
ExecRemote ....................................................................... 455
Exp ................................................................................... 458
ExpandAll .......................................................................... 459
ExpandItem ....................................................................... 459
Fact .................................................................................. 460
FileClose .......................................................................... 460
FileCopy ............................................................................ 461
FileDelete .......................................................................... 462
FileEncoding ..................................................................... 463
FileExists .......................................................................... 464
FileLength .......................................................................... 465
FileLength64 ..................................................................... 466
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileMove</td>
<td>467</td>
</tr>
<tr>
<td>FileOpen</td>
<td>468</td>
</tr>
<tr>
<td>FileRead</td>
<td>471</td>
</tr>
<tr>
<td>FileReadEx</td>
<td>473</td>
</tr>
<tr>
<td>FileSeek</td>
<td>475</td>
</tr>
<tr>
<td>FileSeek64</td>
<td>477</td>
</tr>
<tr>
<td>FileWrite</td>
<td>478</td>
</tr>
<tr>
<td>FileWriteEx</td>
<td>480</td>
</tr>
<tr>
<td>Fill</td>
<td>482</td>
</tr>
<tr>
<td>FillA</td>
<td>483</td>
</tr>
<tr>
<td>FillW</td>
<td>483</td>
</tr>
<tr>
<td>Find</td>
<td>483</td>
</tr>
<tr>
<td>FindCategory</td>
<td>485</td>
</tr>
<tr>
<td>FindClassDefinition</td>
<td>486</td>
</tr>
<tr>
<td>FindFunctionDefinition</td>
<td>487</td>
</tr>
<tr>
<td>FindItem</td>
<td>488</td>
</tr>
<tr>
<td>FindMatchingFunction</td>
<td>494</td>
</tr>
<tr>
<td>FindNext</td>
<td>496</td>
</tr>
<tr>
<td>FindSeries</td>
<td>497</td>
</tr>
<tr>
<td>FindTypeDef</td>
<td>498</td>
</tr>
<tr>
<td>FromAnsi</td>
<td>500</td>
</tr>
<tr>
<td>FromUnicode</td>
<td>501</td>
</tr>
<tr>
<td>GarbageCollect</td>
<td>502</td>
</tr>
<tr>
<td>GarbageCollectGetTimeLimit</td>
<td>502</td>
</tr>
<tr>
<td>GarbageCollectSetTimeLimit</td>
<td>503</td>
</tr>
<tr>
<td>GetActiveSheet</td>
<td>504</td>
</tr>
<tr>
<td>GetAlignment</td>
<td>505</td>
</tr>
<tr>
<td>GetApplication</td>
<td>505</td>
</tr>
<tr>
<td>GetArgElement</td>
<td>506</td>
</tr>
<tr>
<td>GetAutomationNativePointer</td>
<td>507</td>
</tr>
<tr>
<td>GetByte</td>
<td>508</td>
</tr>
<tr>
<td>GetByteArray</td>
<td>509</td>
</tr>
<tr>
<td>GetCertificateLabel</td>
<td>509</td>
</tr>
<tr>
<td>GetChildrenList</td>
<td>511</td>
</tr>
<tr>
<td>GetColumn</td>
<td>513</td>
</tr>
<tr>
<td>GetCommandDDE</td>
<td>514</td>
</tr>
<tr>
<td>GetCommandDDEOrigin</td>
<td>515</td>
</tr>
<tr>
<td>GetCommandString</td>
<td>516</td>
</tr>
<tr>
<td>GetCompanyName</td>
<td>517</td>
</tr>
<tr>
<td>GetContextKeywords</td>
<td>518</td>
</tr>
<tr>
<td>GetContextService</td>
<td>519</td>
</tr>
<tr>
<td>GetCredentialAttribute</td>
<td>520</td>
</tr>
<tr>
<td>GetCurrentDirectory</td>
<td>523</td>
</tr>
<tr>
<td>GetData</td>
<td>523</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>GetDataDDE</td>
<td>528</td>
</tr>
<tr>
<td>GetDataDDEOrigin</td>
<td>528</td>
</tr>
<tr>
<td>GetDataLabelling</td>
<td>530</td>
</tr>
<tr>
<td>GetDataPieExplode</td>
<td>531</td>
</tr>
<tr>
<td>GetDataStyle</td>
<td>532</td>
</tr>
<tr>
<td>GetDataTransparency</td>
<td>538</td>
</tr>
<tr>
<td>GetDataValue</td>
<td>539</td>
</tr>
<tr>
<td>GetDateLimits</td>
<td>541</td>
</tr>
<tr>
<td>GetDbmlsyncPath</td>
<td>542</td>
</tr>
<tr>
<td>GetDisplayRange</td>
<td>543</td>
</tr>
<tr>
<td>GetDynamicDate</td>
<td>544</td>
</tr>
<tr>
<td>GetDynamicDateTime</td>
<td>546</td>
</tr>
<tr>
<td>GetDynamicDecimal</td>
<td>548</td>
</tr>
<tr>
<td>GetDynamicNumber</td>
<td>549</td>
</tr>
<tr>
<td>GetDynamicString</td>
<td>550</td>
</tr>
<tr>
<td>GetDynamicTime</td>
<td>551</td>
</tr>
<tr>
<td>GetEnvironment</td>
<td>552</td>
</tr>
<tr>
<td>GetFileOpenName</td>
<td>553</td>
</tr>
<tr>
<td>GetFileSaveName</td>
<td>558</td>
</tr>
<tr>
<td>GetFirstSheet</td>
<td>560</td>
</tr>
<tr>
<td>GetFixesVersion</td>
<td>561</td>
</tr>
<tr>
<td>GetFocus</td>
<td>562</td>
</tr>
<tr>
<td>GetFolder</td>
<td>563</td>
</tr>
<tr>
<td>GetGlobalProperty</td>
<td>564</td>
</tr>
<tr>
<td>GetHostObject</td>
<td>565</td>
</tr>
<tr>
<td>GetItem</td>
<td>566</td>
</tr>
<tr>
<td>GetItemAtPointer</td>
<td>569</td>
</tr>
<tr>
<td>GetLastReturn</td>
<td>569</td>
</tr>
<tr>
<td>GetLibraryList</td>
<td>570</td>
</tr>
<tr>
<td>GetMajorVersion</td>
<td>571</td>
</tr>
<tr>
<td>GetMessage</td>
<td>572</td>
</tr>
<tr>
<td>GetMinorVersion</td>
<td>573</td>
</tr>
<tr>
<td>GetName</td>
<td>574</td>
</tr>
<tr>
<td>GetNativePointer</td>
<td>575</td>
</tr>
<tr>
<td>GetNextSheet</td>
<td>576</td>
</tr>
<tr>
<td>GetObjectRevisionFromRegistry</td>
<td>577</td>
</tr>
<tr>
<td>GetOrigin</td>
<td>578</td>
</tr>
<tr>
<td>GetParagraphSetting</td>
<td>579</td>
</tr>
<tr>
<td>GetParent</td>
<td>579</td>
</tr>
<tr>
<td>GetPin</td>
<td>581</td>
</tr>
<tr>
<td>GetRecordSet</td>
<td>583</td>
</tr>
<tr>
<td>GetRemote</td>
<td>584</td>
</tr>
<tr>
<td>GetSelectedDate</td>
<td>587</td>
</tr>
<tr>
<td>GetSelectedRange</td>
<td>588</td>
</tr>
</tbody>
</table>
Contents

GetSeriesLabelling ................................................................. 589
GetSeriesStyle ............................................................... 590
GetSeriesTransparency ....................................................... 597
GetShortName .................................................................. 598
GetSpacing ....................................................................... 599
GetStatus ......................................................................... 599
GetSyncRegistryProperties ............................................... 601
GetText .............................................................................. 602
GetTextColor ...................................................................... 603
GetTextStyle ...................................................................... 604
GetToday ............................................................................ 605
GetToolbar ......................................................................... 605
GetToolbarPos .................................................................... 607
GetTransactionName .......................................................... 610
GetURL .............................................................................. 611
GetValue ............................................................................. 612
GetVersionName ............................................................... 613
Handle ................................................................................ 614
Hide .................................................................................. 616
Hour .................................................................................. 617
HyperLinkToURL ............................................................... 617
Idle ................................................................................... 618
ImpersonateClient .............................................................. 619
ImportClipboard .................................................................. 620
ImportFile .......................................................................... 623
ImportString ...................................................................... 627
IncomingCallList ................................................................. 629
Init .................................................................................... 631
InputFieldChangeData ........................................................ 634
InputFieldCurrentName ...................................................... 635
InputFieldDeleteCurrent ..................................................... 636
InputFieldGetData ............................................................... 637
InputFieldInsert ................................................................. 638
InputFieldLocate ............................................................... 638
InsertCategory ................................................................. 640
InsertClass ......................................................................... 642
InsertColumn ...................................................................... 642
InsertData .......................................................................... 643
InsertDocument .................................................................. 646
InsertFile ............................................................................ 647
InsertItem ........................................................................... 648
InsertItemFirst ................................................................. 654
InsertItemLast .................................................................... 656
InsertItemSort .................................................................... 659
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertObject</td>
<td>661</td>
</tr>
<tr>
<td>InsertPicture</td>
<td>662</td>
</tr>
<tr>
<td>InsertSeries</td>
<td>663</td>
</tr>
<tr>
<td>Int</td>
<td>664</td>
</tr>
<tr>
<td>Integer</td>
<td>664</td>
</tr>
<tr>
<td>InternetData</td>
<td>666</td>
</tr>
<tr>
<td>IntHigh</td>
<td>667</td>
</tr>
<tr>
<td>IntLow</td>
<td>667</td>
</tr>
<tr>
<td>InvokePBFuntion</td>
<td>668</td>
</tr>
<tr>
<td>_Is_A</td>
<td>670</td>
</tr>
<tr>
<td>IsAlive</td>
<td>671</td>
</tr>
<tr>
<td>IsAllArabic</td>
<td>672</td>
</tr>
<tr>
<td>IsAllHebrew</td>
<td>672</td>
</tr>
<tr>
<td>IsAnyArabic</td>
<td>673</td>
</tr>
<tr>
<td>IsAnyHebrew</td>
<td>674</td>
</tr>
<tr>
<td>IsArabic</td>
<td>674</td>
</tr>
<tr>
<td>IsArabicAndNumbers</td>
<td>675</td>
</tr>
<tr>
<td>IsCallerInRole</td>
<td>676</td>
</tr>
<tr>
<td>IsDate</td>
<td>678</td>
</tr>
<tr>
<td>IsHebrew</td>
<td>678</td>
</tr>
<tr>
<td>IsHebrewAndNumbers</td>
<td>679</td>
</tr>
<tr>
<td>IsImpersonating</td>
<td>680</td>
</tr>
<tr>
<td>IsInTransaction</td>
<td>681</td>
</tr>
<tr>
<td>IsNull</td>
<td>682</td>
</tr>
<tr>
<td>IsNumber</td>
<td>683</td>
</tr>
<tr>
<td>IsPreview</td>
<td>684</td>
</tr>
<tr>
<td>IsSecurityEnabled</td>
<td>685</td>
</tr>
<tr>
<td>IsTime</td>
<td>686</td>
</tr>
<tr>
<td>IsTransactionAborted</td>
<td>687</td>
</tr>
<tr>
<td>IsValid</td>
<td>688</td>
</tr>
<tr>
<td>KeyDown</td>
<td>689</td>
</tr>
<tr>
<td>LastPos</td>
<td>692</td>
</tr>
<tr>
<td>Left</td>
<td>694</td>
</tr>
<tr>
<td>LeftA</td>
<td>694</td>
</tr>
<tr>
<td>LeftW</td>
<td>695</td>
</tr>
<tr>
<td>LeftTrim</td>
<td>695</td>
</tr>
<tr>
<td>LeftTrimW</td>
<td>696</td>
</tr>
<tr>
<td>Len</td>
<td>697</td>
</tr>
<tr>
<td>LenA</td>
<td>698</td>
</tr>
<tr>
<td>LenW</td>
<td>699</td>
</tr>
<tr>
<td>Length</td>
<td>699</td>
</tr>
<tr>
<td>LibraryCreate</td>
<td>700</td>
</tr>
<tr>
<td>LibraryDelete</td>
<td>701</td>
</tr>
<tr>
<td>LibraryDirectory</td>
<td>701</td>
</tr>
</tbody>
</table>
Contents

LibraryDirectoryEx ................................................................. 703
LibraryExport ................................................................. 705
LibraryImport ................................................................. 706
LineCount .............................................................................. 708
LineLength .............................................................................. 709
LineList .................................................................................. 710
LinkTo ................................................................................... 711
LoadInk .................................................................................. 712
LoadPicture ............................................................................. 713
Log .......................................................................................... 714
LogTen .................................................................................... 715
Long ......................................................................................... 716
LongLong ............................................................................... 718
Lookup ..................................................................................... 720
Lower ....................................................................................... 725
LowerBound ............................................................................. 725
mailAddress ........................................................................... 726
mailDeleteMessage ................................................................ 728
mailGetMessages .................................................................. 729
mailHandle .............................................................................. 731
mailLogoff .............................................................................. 731
mailLogon ............................................................................... 732
mailReadMessage .................................................................. 734
mailRecipientDetails ......................................................... 736
mailResolveRecipient ......................................................... 737
mailSaveMessage ............................................................... 739
mailSend ................................................................................ 742
Match ....................................................................................... 743
MatchW ................................................................................... 746
Max ......................................................................................... 747
MemberDelete ......................................................................... 748
MemberExists ......................................................................... 749
MemberRename ......................................................................... 750
MessageBox ............................................................................ 751
Mid .......................................................................................... 753
MidA ......................................................................................... 755
MidW ......................................................................................... 755
Min .......................................................................................... 756
Minute ...................................................................................... 756
Mod .......................................................................................... 757
ModifyData ............................................................................... 757
Month ....................................................................................... 759
Move ........................................................................................ 760
MoveTab .................................................................................. 761
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Narrow</td>
<td>762</td>
</tr>
<tr>
<td>NextActivity</td>
<td>763</td>
</tr>
<tr>
<td>Now</td>
<td>765</td>
</tr>
<tr>
<td>ObjectAtPointer</td>
<td>765</td>
</tr>
<tr>
<td>Object_To_String</td>
<td>768</td>
</tr>
<tr>
<td>OffsetPos</td>
<td>789</td>
</tr>
<tr>
<td>Open</td>
<td>769</td>
</tr>
<tr>
<td>OpenChannel</td>
<td>785</td>
</tr>
<tr>
<td>OpenSheet</td>
<td>787</td>
</tr>
<tr>
<td>OpenSheetWithParm</td>
<td>789</td>
</tr>
<tr>
<td>OpenTab</td>
<td>793</td>
</tr>
<tr>
<td>OpenTabWithParm</td>
<td>796</td>
</tr>
<tr>
<td>OpenUserObject</td>
<td>801</td>
</tr>
<tr>
<td>OpenUserObjectWithParm</td>
<td>805</td>
</tr>
<tr>
<td>OpenWithParm</td>
<td>809</td>
</tr>
<tr>
<td>OutgoingCallList</td>
<td>814</td>
</tr>
<tr>
<td>PageCount</td>
<td>816</td>
</tr>
<tr>
<td>PageCreated</td>
<td>816</td>
</tr>
<tr>
<td>ParentWindow</td>
<td>817</td>
</tr>
<tr>
<td>Paste</td>
<td>818</td>
</tr>
<tr>
<td>PasteLink</td>
<td>820</td>
</tr>
<tr>
<td>PasteRTF</td>
<td>821</td>
</tr>
<tr>
<td>PasteSpecial</td>
<td>822</td>
</tr>
<tr>
<td>PBGetMenuString</td>
<td>823</td>
</tr>
<tr>
<td>Pi</td>
<td>824</td>
</tr>
<tr>
<td>PixelsToUnits</td>
<td>824</td>
</tr>
<tr>
<td>Play</td>
<td>825</td>
</tr>
<tr>
<td>PointerX</td>
<td>826</td>
</tr>
<tr>
<td>PointerY</td>
<td>827</td>
</tr>
<tr>
<td>PopMenu</td>
<td>828</td>
</tr>
<tr>
<td>PopulateError</td>
<td>829</td>
</tr>
<tr>
<td>Pos</td>
<td>830</td>
</tr>
<tr>
<td>PosA</td>
<td>832</td>
</tr>
<tr>
<td>PosW</td>
<td>832</td>
</tr>
<tr>
<td>Position</td>
<td>832</td>
</tr>
<tr>
<td>Post</td>
<td>838</td>
</tr>
<tr>
<td>PostEvent</td>
<td>839</td>
</tr>
<tr>
<td>PostURL</td>
<td>841</td>
</tr>
<tr>
<td>Preview</td>
<td>843</td>
</tr>
<tr>
<td>Print</td>
<td>845</td>
</tr>
<tr>
<td>PrintBitmap</td>
<td>851</td>
</tr>
<tr>
<td>PrintCancel</td>
<td>852</td>
</tr>
<tr>
<td>PrintClose</td>
<td>854</td>
</tr>
<tr>
<td>PrintDataWindow</td>
<td>855</td>
</tr>
</tbody>
</table>
### Contents

- PrintDefineFont ................................................................. 856
- PrintEx ................................................................. 857
- PrintGetPrinter .......................................................... 858
- PrintGetPrinters ............................................................. 859
- PrintLine ................................................................. 860
- PrintOpen ................................................................. 861
- PrintOval ................................................................. 862
- PrintPage ................................................................. 864
- PrintRect ................................................................. 865
- PrintRoundRect ............................................................ 866
- PrintScreen ............................................................... 867
- PrintSend ................................................................. 868
- PrintSetFont ............................................................... 870
- PrintSetPrinter ............................................................ 871
- PrintSetSpacing ............................................................ 872
- PrintSetup ............................................................... 873
- PrintSetupPrinter ......................................................... 873
- PrintText ................................................................. 874
- PrintWidth ................................................................. 876
- PrintX ................................................................. 877
- PrintY ................................................................. 877
- ProfileInt ................................................................. 878
- ProfileString .............................................................. 880
- Rand ................................................................. 881
- Randomize ............................................................... 882
- Read ................................................................. 883
- Real ................................................................. 886
- RecognizeText ............................................................. 887
- RegistryDelete ............................................................. 887
- RegistryGet ............................................................... 888
- RegistryKeys ............................................................... 889
- RegistrySet ............................................................... 891
- RegistryValues ......................................................... 893
- RelativeDate .............................................................. 893
- RelativeTime .............................................................. 894
- ReleaseAutomationNativePointer ......................... 895
- ReleaseNativePointer .................................................. 896
- RemoveDirectory .......................................................... 896
- Repair ................................................................. 897
- Replace ................................................................. 898
- ReplaceA ................................................................. 900
- ReplaceText ............................................................... 901
- ReplaceW ................................................................. 902
- Reset ................................................................. 902
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResetArgElements</td>
<td>905</td>
</tr>
<tr>
<td>ResetDataColors</td>
<td>906</td>
</tr>
<tr>
<td>ResetInk</td>
<td>907</td>
</tr>
<tr>
<td>ResetPicture</td>
<td>908</td>
</tr>
<tr>
<td>Resolve_Initial_References</td>
<td>909</td>
</tr>
<tr>
<td>RespondRemote</td>
<td>911</td>
</tr>
<tr>
<td>Restart</td>
<td>912</td>
</tr>
<tr>
<td>ResumeTransaction</td>
<td>912</td>
</tr>
<tr>
<td>Reverse</td>
<td>914</td>
</tr>
<tr>
<td>RevertToSelf</td>
<td>915</td>
</tr>
<tr>
<td>RGB</td>
<td>916</td>
</tr>
<tr>
<td>Right</td>
<td>917</td>
</tr>
<tr>
<td>RightA</td>
<td>918</td>
</tr>
<tr>
<td>RightW</td>
<td>919</td>
</tr>
<tr>
<td>RightTrim</td>
<td>919</td>
</tr>
<tr>
<td>RightTrimW</td>
<td>920</td>
</tr>
<tr>
<td>RollbackOnly</td>
<td>920</td>
</tr>
<tr>
<td>RollbackTransaction</td>
<td>921</td>
</tr>
<tr>
<td>Round</td>
<td>923</td>
</tr>
<tr>
<td>RoutineList</td>
<td>924</td>
</tr>
<tr>
<td>Run</td>
<td>925</td>
</tr>
<tr>
<td>Save</td>
<td>927</td>
</tr>
<tr>
<td>SaveAs</td>
<td>930</td>
</tr>
<tr>
<td>SaveDocument</td>
<td>938</td>
</tr>
<tr>
<td>SaveInk</td>
<td>940</td>
</tr>
<tr>
<td>Scroll</td>
<td>942</td>
</tr>
<tr>
<td>ScrollNextPage</td>
<td>943</td>
</tr>
<tr>
<td>ScrollNextRow</td>
<td>944</td>
</tr>
<tr>
<td>ScrollPriorPage</td>
<td>945</td>
</tr>
<tr>
<td>ScrollPriorRow</td>
<td>946</td>
</tr>
<tr>
<td>ScrollToRow</td>
<td>947</td>
</tr>
<tr>
<td>Second</td>
<td>948</td>
</tr>
<tr>
<td>SecondsAfter</td>
<td>949</td>
</tr>
<tr>
<td>Seek</td>
<td>950</td>
</tr>
<tr>
<td>SelectedColumn</td>
<td>952</td>
</tr>
<tr>
<td>SelectedIndex</td>
<td>953</td>
</tr>
<tr>
<td>SelectedItem</td>
<td>953</td>
</tr>
<tr>
<td>SelectedLength</td>
<td>954</td>
</tr>
<tr>
<td>SelectedLine</td>
<td>956</td>
</tr>
<tr>
<td>SelectedPage</td>
<td>957</td>
</tr>
<tr>
<td>SelectedStart</td>
<td>958</td>
</tr>
<tr>
<td>SelectedText</td>
<td>959</td>
</tr>
<tr>
<td>SelectionRange</td>
<td>960</td>
</tr>
</tbody>
</table>
Contents

SelectItem ................................................................. 961
SelectObject ............................................................. 964
SelectTab ................................................................. 965
SelectText ................................................................. 966
SelectTextAll ........................................................... 970
SelectTextLine .......................................................... 971
SelectTextWord ......................................................... 971
Send ........................................................................ 973
SeriesCount ............................................................. 975
SeriesName .............................................................. 976
SetAbort ................................................................... 977
SetAlignment ............................................................ 979
SetArgElement .......................................................... 980
SetAutomationLocale .................................................. 981
SetAutomationPointer ............................................... 982
SetAutomationTimeout .............................................. 983
SetBoldDate .............................................................. 985
SetByte ................................................................... 986
SetColumn ............................................................... 987
SetComplete ............................................................. 988
SetData ................................................................. 991
SetDataDDE .............................................................. 992
SetDataLabelling ....................................................... 993
SetDataPieExplode ..................................................... 995
SetDataStyle ............................................................. 996
SetDataTransparency .................................................. 1002
SetDateLimits ........................................................... 1004
SetDropHighlight ....................................................... 1004
SetDynamicParm ....................................................... 1005
SetFirstVisible .......................................................... 1006
SetFocus ................................................................ 1007
SetGlobalProperty ..................................................... 1008
SetItem .................................................................. 1009
SetLevelPictures ....................................................... 1012
SetLibraryList .......................................................... 1013
SetMask .................................................................. 1015
SetMessage ............................................................. 1016
SetMicroHelp ............................................................ 1017
SetNewMobiLinkPassword .......................................... 1018
SetNull .................................................................. 1019
SetOverlayPicture ...................................................... 1020
SetParagraphSetting .................................................. 1021
SetParm ................................................................. 1022
SetPicture ............................................................... 1023
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetPointer</td>
<td>1024</td>
</tr>
<tr>
<td>SetPosition</td>
<td>1026</td>
</tr>
<tr>
<td>SetProfileString</td>
<td>1028</td>
</tr>
<tr>
<td>SetRange</td>
<td>1030</td>
</tr>
<tr>
<td>SetRecordSet</td>
<td>1030</td>
</tr>
<tr>
<td>SetRedraw</td>
<td>1032</td>
</tr>
<tr>
<td>SetRemote</td>
<td>1032</td>
</tr>
<tr>
<td>SetResultSet</td>
<td>1035</td>
</tr>
<tr>
<td>SetSelectedDate</td>
<td>1036</td>
</tr>
<tr>
<td>SetSelectedRange</td>
<td>1037</td>
</tr>
<tr>
<td>SetSeriesLabelling</td>
<td>1038</td>
</tr>
<tr>
<td>SetSeriesStyle</td>
<td>1039</td>
</tr>
<tr>
<td>SetSeriesTransparency</td>
<td>1046</td>
</tr>
<tr>
<td>SetSpacing</td>
<td>1048</td>
</tr>
<tr>
<td>SetState</td>
<td>1049</td>
</tr>
<tr>
<td>SetSyncRegistryProperties</td>
<td>1050</td>
</tr>
<tr>
<td>SetTextColor</td>
<td>1051</td>
</tr>
<tr>
<td>SetTextStyle</td>
<td>1051</td>
</tr>
<tr>
<td>SetTimeout</td>
<td>1052</td>
</tr>
<tr>
<td>SetToday</td>
<td>1054</td>
</tr>
<tr>
<td>SetToolbar</td>
<td>1055</td>
</tr>
<tr>
<td>SetToolbarPos</td>
<td>1057</td>
</tr>
<tr>
<td>SetTop</td>
<td>1061</td>
</tr>
<tr>
<td>SetTraceFileName</td>
<td>1061</td>
</tr>
<tr>
<td>SetTransPool</td>
<td>1063</td>
</tr>
<tr>
<td>SetValue</td>
<td>1063</td>
</tr>
<tr>
<td>SharedObjectDirectory</td>
<td>1065</td>
</tr>
<tr>
<td>SharedObjectGet</td>
<td>1065</td>
</tr>
<tr>
<td>SharedObjectRegister</td>
<td>1069</td>
</tr>
<tr>
<td>SharedObjectUnregister</td>
<td>1070</td>
</tr>
<tr>
<td>Show</td>
<td>1070</td>
</tr>
<tr>
<td>ShowHeadFoot</td>
<td>1071</td>
</tr>
<tr>
<td>ShowHelp</td>
<td>1072</td>
</tr>
<tr>
<td>ShowPopupHelp</td>
<td>1074</td>
</tr>
<tr>
<td>Sign</td>
<td>1075</td>
</tr>
<tr>
<td>SignalError</td>
<td>1075</td>
</tr>
<tr>
<td>Sin</td>
<td>1076</td>
</tr>
<tr>
<td>Sleep</td>
<td>1077</td>
</tr>
<tr>
<td>Sort</td>
<td>1077</td>
</tr>
<tr>
<td>SortAll</td>
<td>1079</td>
</tr>
<tr>
<td>Space</td>
<td>1080</td>
</tr>
<tr>
<td>SQRT</td>
<td>1081</td>
</tr>
<tr>
<td>Start</td>
<td>1082</td>
</tr>
<tr>
<td>StartHotLink</td>
<td>1089</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
</tr>
<tr>
<td>UpdateLinksDialog</td>
<td>1150</td>
</tr>
<tr>
<td>Upper</td>
<td>1152</td>
</tr>
<tr>
<td>UpperBound</td>
<td>1152</td>
</tr>
<tr>
<td>Which</td>
<td>1154</td>
</tr>
<tr>
<td>WordCap</td>
<td>1156</td>
</tr>
<tr>
<td>WorkSpaceHeight</td>
<td>1156</td>
</tr>
<tr>
<td>WorkSpaceWidth</td>
<td>1158</td>
</tr>
<tr>
<td>WorkSpaceX</td>
<td>1158</td>
</tr>
<tr>
<td>WorkSpaceY</td>
<td>1159</td>
</tr>
<tr>
<td>Write</td>
<td>1160</td>
</tr>
<tr>
<td>XMLParseFile</td>
<td>1161</td>
</tr>
<tr>
<td>XMLParseString</td>
<td>1165</td>
</tr>
<tr>
<td>Year</td>
<td>1168</td>
</tr>
<tr>
<td>Yield</td>
<td>1169</td>
</tr>
</tbody>
</table>

Index ......................................................................................................................................... 1173
## About This Book

<table>
<thead>
<tr>
<th>Audience</th>
<th>This book is for programmers who will use PowerBuilder® to build client/server or multitier applications.</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to use this book</td>
<td>This book describes syntax and usage information for the PowerScript® language including variables, expressions, statements, events, and functions.</td>
</tr>
<tr>
<td>Related documents</td>
<td>For a complete list of PowerBuilder documentation, see the preface of the PowerBuilder <em>Getting Started</em> book.</td>
</tr>
<tr>
<td>Other sources of information</td>
<td>Use the Sybase® Getting Started CD, the SyBooks™ CD, and the Sybase Product Manuals Web site to learn more about your product:</td>
</tr>
<tr>
<td></td>
<td>• The <em>Getting Started</em> CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the <em>Getting Started</em> CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.</td>
</tr>
<tr>
<td></td>
<td>• The <em>SyBooks</em> CD contains product manuals and is included with your software. The Eclipse-based <em>SyBooks</em> browser allows you to access the manuals in an easy-to-use, HTML-based format. Some documentation may be provided in PDF format, which you can access through the PDF directory on the <em>SyBooks</em> CD. To read or print the PDF files, you need Adobe Acrobat Reader.</td>
</tr>
<tr>
<td></td>
<td>Refer to the <em>SyBooks Installation Guide</em> on the <em>Getting Started</em> CD, or the README.txt file on the <em>SyBooks</em> CD for instructions on installing and starting <em>SyBooks</em>.</td>
</tr>
<tr>
<td></td>
<td>• The <em>Sybase Product Manuals</em> Web site is an online version of the <em>SyBooks</em> CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the <em>Sybase Developer Network</em>. To access the <em>Sybase Product Manuals</em> Web site, go to Product Manuals at <a href="http://www.sybase.com/support/manuals/">http://www.sybase.com/support/manuals/</a>.</td>
</tr>
</tbody>
</table>
### Conventions

The formatting conventions used in this manual are:

<table>
<thead>
<tr>
<th>Formatting example</th>
<th>Indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Retrieve and Update</strong></td>
<td>When used in descriptive text, this font indicates:</td>
</tr>
<tr>
<td></td>
<td>- Command, function, and method names</td>
</tr>
<tr>
<td></td>
<td>- Keywords such as true, false, and null</td>
</tr>
<tr>
<td></td>
<td>- Datatypes such as integer and char</td>
</tr>
<tr>
<td></td>
<td>- Database column names such as emp_id and l_name</td>
</tr>
<tr>
<td></td>
<td>- User-defined objects such as dw_emp or w_main</td>
</tr>
<tr>
<td><strong>variable or file name</strong></td>
<td>When used in descriptive text and syntax descriptions, oblique font indicates:</td>
</tr>
<tr>
<td></td>
<td>- Variables, such as <em>myCounter</em></td>
</tr>
<tr>
<td></td>
<td>- Parts of input text that must be substituted, such as <em>pdbname.pbd</em></td>
</tr>
<tr>
<td></td>
<td>- File and path names</td>
</tr>
<tr>
<td><strong>File&gt;Save</strong></td>
<td>Menu names and menu items are displayed in plain text. The greater than symbol (&gt;) shows you how to navigate menu selections. For example, File&gt;Save indicates “select Save from the File menu.”</td>
</tr>
<tr>
<td><strong>dw_emp.Update()</strong></td>
<td>Monospace font indicates:</td>
</tr>
<tr>
<td></td>
<td>- Information that you enter in a dialog box or on a command line</td>
</tr>
<tr>
<td></td>
<td>- Sample script fragments</td>
</tr>
<tr>
<td></td>
<td>- Sample output fragments</td>
</tr>
</tbody>
</table>

### If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.
PART 1

PowerScript Topics
CHAPTER 1

Language Basics

About this chapter

This chapter describes general elements and conventions of PowerScript.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>3</td>
</tr>
<tr>
<td>Identifier names</td>
<td>5</td>
</tr>
<tr>
<td>Labels</td>
<td>6</td>
</tr>
<tr>
<td>Special ASCII characters</td>
<td>6</td>
</tr>
<tr>
<td>NULL values</td>
<td>8</td>
</tr>
<tr>
<td>Reserved words</td>
<td>10</td>
</tr>
<tr>
<td>Pronouns</td>
<td>11</td>
</tr>
<tr>
<td>Statement continuation</td>
<td>15</td>
</tr>
<tr>
<td>Statement separation</td>
<td>16</td>
</tr>
<tr>
<td>White space</td>
<td>16</td>
</tr>
<tr>
<td>Conditional compilation</td>
<td>17</td>
</tr>
</tbody>
</table>

Comments

Description

You can use comments to document your scripts and prevent statements within a script from executing. There are two methods.

Syntax

Double-slash method

```
Code // Comment
```

Slash-and-asterisk method

```
/* Comment */
```
Comments

Usage

The following table shows how to use each method.

**Table 1-1: Methods for adding comments in scripts**

<table>
<thead>
<tr>
<th>Method</th>
<th>Marker</th>
<th>Can use to</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double slash</td>
<td>//</td>
<td>Designate all text on the line to the right of</td>
<td>Cannot extend to multiple lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the marker as a comment</td>
<td></td>
</tr>
<tr>
<td>Slash and</td>
<td>/<em>...</em>/</td>
<td>Designate the text between the markers as a</td>
<td>• Can extend over multiple lines (multiline comments do not require a continuation character)</td>
</tr>
<tr>
<td>asterisk</td>
<td></td>
<td>comment</td>
<td>• Can be nested</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nest comments</td>
<td></td>
</tr>
</tbody>
</table>

Adding comment markers

In Script views and the Function painter, you can use the Comment Selection button (or select Edit>Comment Selection from the menu bar) to comment out the line containing the cursor or a selected group of lines.

For information about adding comments to objects and library entries, see the *PowerBuilder Users Guide*.

Examples

**Double-slash method**

// This entire line is a comment.
// This entire line is another comment.
amt = qty * cost // Rest of the line is comment.

// The following statement was commented out so that it // would not execute.
// SetNull(amt)

**Slash-and-asterisk method**

/* This is a single-line comment. */

/* This comment starts here, continues to this line, and finally ends here. */

A = B + C /* This comment starts here.
/* This is the start of a nested comment.
The nested comment ends here. */
The first comment ends here. */ + D + E + F
Identifier names

You use identifiers to name variables, labels, functions, windows, controls, menus, and anything else you refer to in scripts.

Syntax

Rules for identifiers:

- Must start with a letter or an _ (underscore)
- Cannot be reserved words (see “Reserved words” on page 10)
- Can have up to 40 characters but no spaces
- Are not case sensitive (PART, Part, and part are identical)
- Can include any combination of letters, numbers, and these special characters:

  - Dash
  - Underscore
  - Dollar sign
  - Number sign
  - Percent sign

Usage

By default, PowerBuilder allows you to use dashes in all identifiers, including in variable names in a script. However, this means that when you use the subtraction operator or the -- operator in a script, you must surround it with spaces. If you do not, PowerBuilder interprets the expression as an identifier name.

If you want to disallow dashes in variable names in scripts, you can change the setting of the Allow Dashes in Identifiers option in the script editor’s property sheet. As a result, you do not have to surround the subtraction operator and the decrement assignment shortcut (--) with spaces.

Be careful

If you disallow dashes and have previously used dashes in variable names, you will get errors the next time you compile.

Examples

Valid identifiers

- ABC_Code
- Child-Id
- FirstButton
- response35
- pay-before%deductions$
- ORDER_DATE
Labels

Description
You can include labels in scripts for use with GOTO statements.

Syntax
Identifier.

Usage
A label can be any valid identifier. You can enter it on a line by itself above the statement or at the start of the line before the statement.

For information about the GOTO statement, see GOTO on page 136. For information about valid identifiers, see “Identifier names” on page 5.

Examples

On a line by itself above the statement

FindCity:
IF city=cityname[1] THEN ...

At the start of the line before the statement

FindCity: IF city=cityname[1] THEN ...

Special ASCII characters

Description
You can include special ASCII characters in strings. For example, you might want to include a tab in a string to ensure proper spacing or a bullet to indicate a list item. The tilde character (~) introduces special characters. The tab is one of the common ASCII characters that can be entered by typing a tilde followed by a single keystroke. The bullet must be entered by typing a tilde followed by the decimal, hexadecimal, or octal ASCII value that represents it.
Follow the guidelines in the following table.

**Table 1-2: Using special ASCII characters in strings**

<table>
<thead>
<tr>
<th>In this category</th>
<th>To specify this</th>
<th>Enter this</th>
<th>More information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common ASCII characters</td>
<td>Newline</td>
<td>~n</td>
<td></td>
</tr>
<tr>
<td>Tab</td>
<td>~t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertical tab</td>
<td>~v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carriage return</td>
<td>~r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form feed</td>
<td>~f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backspace</td>
<td>~b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double quote</td>
<td>~&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single quote</td>
<td>~'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tilde</td>
<td>~~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Any ASCII character</td>
<td>Decimal</td>
<td>~###</td>
<td>### = a 3-digit number from 000 to 255</td>
</tr>
<tr>
<td></td>
<td>Hexadecimal</td>
<td>~h##</td>
<td>## = a 2-digit hexadecimal number from 01 to FF</td>
</tr>
<tr>
<td></td>
<td>Octal</td>
<td>~o###</td>
<td>### = a 3-digit octal number from 000 to 377</td>
</tr>
</tbody>
</table>

**Examples**

**Entering ASCII characters** Here is how to use special characters in strings:

<table>
<thead>
<tr>
<th>String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;dog~n&quot;</td>
<td>A string containing the word dog followed by a newline character</td>
</tr>
<tr>
<td>&quot;dog<del>tcat</del>ttiger&quot;</td>
<td>A string containing the word dog, a tab character, the word cat, another tab character, and the word tiger</td>
</tr>
</tbody>
</table>

**Using decimal, hexadecimal, and octal values** Here is how to indicate a bullet (•) in a string by using the decimal, hexadecimal, and octal ASCII values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;~249&quot;</td>
<td>The ASCII character with decimal value 249</td>
</tr>
<tr>
<td>&quot;~hF9&quot;</td>
<td>The ASCII character with hexadecimal value F9</td>
</tr>
<tr>
<td>&quot;~o371&quot;</td>
<td>The ASCII character with octal value 371</td>
</tr>
</tbody>
</table>
NULL values

Description
Null means *undefined or unknown*. It is not the same as an empty string or zero or a date of 0000-00-00. For example, null is neither 0 nor not 0.

Typically, you work with null values only with respect to database values.

Usage

**Initial values for variables**  Although PowerBuilder supports null values for all variable datatypes, it does not initialize variables to null. Instead, when a variable is not set to a specific value when it is declared, PowerBuilder sets it to the default initial value for the datatype—for example, zero for a numeric value, false for boolean, and the empty string (""") for a string.

**Null variables**  A variable can become null if one of the following occurs:

- A null value is read into it from the database. If your database supports null, and a SQL INSERT or UPDATE statement sends a null to the database, it is written to the database as null and can be read into a variable by a SELECT or FETCH statement.

**Null in a variable**  When a null value is read into a variable, the variable remains null unless it is changed in a script.

- The `SetNull` function is used in a script to set the variable explicitly to null. For example:

  ```
  string city       // city is an empty string.
  SetNull(city)    // city is set to NULL.
  ```

**Nulls in functions and expressions**  Most functions that have a null value for any argument return null. Any expression that has a variable with a null value results in null.

A boolean expression that is null is considered undefined and therefore false.

**Testing for null**  To test whether a variable or expression is null, use the `IsNull` function. You cannot use an equal sign (=) to test for null.

**Valid**  This statement shows the correct way to test for null:

```
IF IsNull(a) THEN ... 
```

**Invalid**  This statement shows the incorrect way to test for null:

```
IF a = NULL THEN ... 
```
Examples

**Example 1** None of the following statements make the computer beep (the variable *nbr* is set to null, so each statement evaluates to false):

```plaintext
int     Nbr
// Set Nbr to NULL.
SetNull(Nbr)
IF Nbr = 1 THEN Beep(1)
IF Nbr <> 1 THEN Beep(1)
IF NOT (Nbr = 1) THEN Beep(1)
```

**Example 2** In this *IF...THEN* statement, the boolean expression evaluates to false, so the *ELSE* is executed:

```plaintext
int     a
SetNull(a)
IF a = 1 THEN
    MessageBox("Value", "a = 1")
ELSE
    MessageBox("Value", "a = NULL")
END IF
```

**Example 3** This example is a more useful application of a null boolean expression than Example 2. It displays a message if no control has focus. When no control has focus, GetFocus returns a null object reference, the boolean expression evaluates to false, and the *ELSE* is executed:

```plaintext
IF GetFocus( ) THEN
    . . . // Some processing
ELSE
    MessageBox("Important", "Specify an option!")
END IF
```
Reserved words

The words PowerBuilder uses internally are called reserved words and cannot be used as identifiers. If you use a reserved word as an identifier, you get a compiler warning. Reserved words that are marked with an asterisk (*) can be used as function names.

Table 1-3: PowerScript reserved words

<table>
<thead>
<tr>
<th>alias</th>
<th>execute</th>
<th>not</th>
<th>step</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>exit</td>
<td>of</td>
<td>super</td>
</tr>
<tr>
<td>autoinstantiate</td>
<td>external</td>
<td>on</td>
<td>system</td>
</tr>
<tr>
<td>call</td>
<td>false</td>
<td>open*</td>
<td>systemread</td>
</tr>
<tr>
<td>case</td>
<td>fetch</td>
<td>or</td>
<td>systemwrite</td>
</tr>
<tr>
<td>catch</td>
<td>finally</td>
<td>parent</td>
<td>systemwrite</td>
</tr>
<tr>
<td>choose</td>
<td>first</td>
<td>post*</td>
<td>then</td>
</tr>
<tr>
<td>close*</td>
<td>for</td>
<td>prepare</td>
<td>this</td>
</tr>
<tr>
<td>commit</td>
<td>forward</td>
<td>prior</td>
<td>throw</td>
</tr>
<tr>
<td>connect</td>
<td>from</td>
<td>private</td>
<td>throws</td>
</tr>
<tr>
<td>constant</td>
<td>function</td>
<td>privateread</td>
<td>to</td>
</tr>
<tr>
<td>continue</td>
<td>global</td>
<td>privatewrite</td>
<td>trigger</td>
</tr>
<tr>
<td>create*</td>
<td>goto</td>
<td>procedure</td>
<td>true</td>
</tr>
<tr>
<td>cursor</td>
<td>halt</td>
<td>protected</td>
<td>try</td>
</tr>
<tr>
<td>declare</td>
<td>if</td>
<td>protectedread</td>
<td>type</td>
</tr>
<tr>
<td>delete</td>
<td>immediate</td>
<td>protectedwrite</td>
<td>until</td>
</tr>
<tr>
<td>describe*</td>
<td>indirect</td>
<td>prototypes</td>
<td>update*</td>
</tr>
<tr>
<td>descriptor</td>
<td>insert</td>
<td>public</td>
<td>updateblob</td>
</tr>
<tr>
<td>destroy</td>
<td>into</td>
<td>readonly</td>
<td>using</td>
</tr>
<tr>
<td>disconnect</td>
<td>intrinsic</td>
<td>ref</td>
<td>variables</td>
</tr>
<tr>
<td>do</td>
<td>is</td>
<td>return</td>
<td>while</td>
</tr>
<tr>
<td>dynamic</td>
<td>last</td>
<td>rollback</td>
<td>with</td>
</tr>
<tr>
<td>else</td>
<td>library</td>
<td>rpcfunc</td>
<td>within</td>
</tr>
<tr>
<td>elseif</td>
<td>loop</td>
<td>select</td>
<td>xor</td>
</tr>
<tr>
<td>end</td>
<td>namespace</td>
<td>selectblob</td>
<td>_debug</td>
</tr>
<tr>
<td>enumerated</td>
<td>native</td>
<td>shared</td>
<td></td>
</tr>
<tr>
<td>event</td>
<td>next</td>
<td>static</td>
<td></td>
</tr>
</tbody>
</table>

The PowerBuilder system class also includes private variables that you cannot use as identifiers. If you use a private variable as an identifier, you get an informational message and should rename your identifier.

If you are deploying a DataWindow to the Web, you cannot use JavaScript reserved words to name fields or bands in the DataWindow object. The list of reserved words is available on the Sun Microsystems Web site at http://docs.sun.com/source/816-6410-10/keywords.htm.
Pronouns

PowerScript has pronouns that allow you to make a general reference to an object or control. When you use a pronoun, the reference remains correct even if the name of the object or control changes.

Usage

You can use pronouns in function and event scripts wherever you would use an object's name. For example, you can use a pronoun to:

- Cause an event in an object or control
- Manipulate or change an object or control
- Obtain or change the setting of a property

The following table lists the PowerScript pronouns and summarizes their use.

<table>
<thead>
<tr>
<th>This pronoun</th>
<th>In a script for a</th>
<th>Refers to the</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>Window, custom user object, menu, application object, or control</td>
<td>Object or control itself</td>
</tr>
<tr>
<td>Parent</td>
<td>Control in a window</td>
<td>Window containing the control</td>
</tr>
<tr>
<td></td>
<td>Control in a custom user object</td>
<td>Custom user object containing the control</td>
</tr>
<tr>
<td></td>
<td>Menu</td>
<td>Item in the menu on the level above the current menu</td>
</tr>
<tr>
<td>Super</td>
<td>Descendent object or control</td>
<td>Parent</td>
</tr>
<tr>
<td></td>
<td>Descendent window or user object</td>
<td>Immediate ancestor of the window or user object</td>
</tr>
<tr>
<td></td>
<td>Control in a descendent window or user object</td>
<td>Immediate ancestor of the control’s parent window or user object</td>
</tr>
</tbody>
</table>

ParentWindow property  You can use the ParentWindow property of the Menu object 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 Users Guide.

The rest of this section describes the individual pronouns in detail.
Pronouns

Parent pronoun

Description
Parent in a PowerBuilder script refers to the object that contains the current object.

Usage
You can use the pronoun Parent in scripts for:
- Controls in windows
- Custom user objects
- Menus

Where you use Parent determines what it references:

Window controls  When you use Parent in a script for a control (such as a CommandButton), Parent refers to the window that contains the control.

User object controls  When you use Parent in a script for a control in a custom user object, Parent refers to the user object.

Menus  When you use Parent in a menu script, Parent refers to the menu item on the level above the menu the script is for.

Examples

Window controls  If you include this statement in the script for the Clicked event in a CommandButton within a window, clicking the button closes the window containing the button:

Close(Parent)

If you include this statement in the script for the CommandButton, clicking the button displays a horizontal scroll bar within the window (sets the HScrollBar property of the window to true):

Parent.HScrollBar = TRUE

User object controls  If you include this statement in a script for the Clicked event for a CheckBox in a user object, clicking the check box hides the user object:

Parent.Hide()

If you include this statement in the script for the CheckBox, clicking the check box disables the user object (sets the Enabled property of the user object to false):

Parent.Enabled = FALSE

Menus  If you include this statement in the script for the Clicked event in the menu item Select All under the menu item Select, clicking Select All disables the menu item Select:

Parent.Disable()
If you include this statement in the script for the Clicked event in the menu item Select All, clicking Select All checks the menu item Select:

```
Parent.Checked = TRUE
```

### This pronoun

**Description**
The pronoun This in a PowerBuilder script refers to the window, user object, menu, application object, or control that owns the current script.

**Usage**

**Why include This**
Using This allows you to make ownership explicit. The following statement refers to the current object’s X property:

```
This.X = This.X + 50
```

**When optional but helpful**
In the script for an object or control, you can refer to the properties of the object or control without qualification, but it is good programming practice to include This to make the script clear and easy to read.

**When required**
There are some circumstances when you must use This. When a global or local variable has the same name as an instance variable, PowerBuilder finds the global or local variable first. Qualifying the variable with This allows you to refer to the instance variable instead of the global variable.

**EAServer restriction**
You cannot use This to pass arguments in EAServer components.

### Examples

**Example 1**
This statement in a script for a menu places a check mark next to the menu selection:

```
This.Check( )
```

**Example 2**
In this function call, This passes a reference to the object containing the script:

```
ReCalc(This)
```

**Example 3**
If you omit This, “x” in the following statement refers to a local variable x if there is one defined (the script adds 50 to the variable x, not to the X property of the control). It refers to the object’s X property if there is no local variable:

```
x = x + 50
```
Example 4  Use This to ensure that you refer to the property. For example, in the following statement in the script for the Clicked event for a CommandButton, clicking the button changes the horizontal position of the button (changes the button’s X property):

\[ \text{This.x} = \text{This.x} + 50 \]

Super pronoun

Description
When you write a PowerBuilder script for a descendant object or control, you can call scripts written for any ancestor. You can directly name the ancestor in the call, or you can use the reserved word Super to refer to the immediate ancestor.

Usage

Whether to use Super  If you are calling an ancestor function, you only need to use Super if the descendant has a function with the same name and the same arguments as the ancestor function. Otherwise, you would simply call the function with no qualifiers.

Restrictions for Super  You cannot use Super to call scripts associated with controls in the ancestor window. You can only use Super in an event or function associated with a direct descendant of the ancestor whose function is being called. Otherwise, the compiler returns a syntax error.

To call scripts associated with controls, use the CALL statement.

See the discussion of CALL on page 123.

Examples

Example 1  This example calls the ancestor function wf_myfunc (presumably the descendant also has a function called wf_myfunc):

\[ \text{Super::wf_myfunc(myarg1, myarg2)} \]

This example must be part of a script or function in the descendant window, not one of the window’s controls. For example, if it is in the Clicked event of a button on the descendant window, you get a syntax error when the script is compiled.

Supplying arguments
Be certain to supply the correct number of arguments for the ancestor function.

Example 2  This example in a CommandButton script calls the Clicked script for the CommandButton in the immediate ancestor window or user object:

\[ \text{Super::EVENT Clicked()} \]
## Statement continuation

**Description**

Although you typically put one statement on each line, you occasionally need to continue a statement to more than one line. The statement continuation character is the ampersand (`&`). (For the use of the ampersand character in accelerator keys, see the PowerBuilder Users Guide.)

**Syntax**

\[
\text{Start of statement \&} \\
\text{more statement \&} \\
\text{end of statement}
\]

The ampersand must be the last nonwhite character on the line or the compiler considers it part of the statement.

For information about white space, see “White space” on page 16.

**Usage**

You do not use a continuation character for:

- **Continuing comments**  *Do not* use a continuation character to continue a comment. The continuation character is considered part of the comment and is ignored by the compiler.

- **Continuing SQL statements**  *You do not* need a continuation character to continue a SQL statement. In PowerBuilder, SQL statements always end with a semicolon (`;`), and the compiler considers everything from the start of a SQL statement to a semicolon to be part of the SQL statement. A continuation character in a SQL statement is considered part of the statement and usually causes an error.

**Examples**

**Continuing a quoted string**

*One way*  Place an ampersand in the middle of the string and continue the string on the next line:

```plaintext
IF Employee_District = "Eastern United States and&
Eastern Canada" THEN ... 
```

Note that any white space (such as tabs and spaces) before the ampersand and at the beginning of the continued line is part of the string.

*A problem*  The following statement uses only the ampersand to continue the quoted string in the IF...THEN statement to another line; for readability, a tab has been added to indent the second line. The compiler includes the tab in the string, which might result in an error:

```plaintext
IF Employee_District = "Eastern United States and&
Eastern Canada" THEN ... 
```
A better way  A better way to continue a quoted string is to enter a quotation mark before the continuation character ("& or ", depending on whether the string is delimited by single or double quotation marks) at the end of the first line of the string and a plus sign and a quotation mark ("+" or "+") at the start of the next line. This way, you do not inadvertently include unwanted characters (such as tabs or spaces) in the string literal:

```powerbuilder
IF Employee_District = "Eastern United States and "&
+" Eastern Canada" THEN ...
```

The examples in the PowerBuilder documentation use this method to continue quoted strings.

Continuing a variable name  Do not split a line by inserting the continuation character within a variable name. This causes an error and the statement fails, because the continuation character splits the variable name “Quantity”:

```powerbuilder
Total-Cost = Price * Quan&
tity + (Tax + Shipping)
```

### Statement separation

**Description**

Although you typically put one statement on each line, you occasionally want to combine multiple statements on a single line. The statement separation character is the semicolon (;).

**Syntax**

```
Statement1; statement2
```

**Examples**

The following line contains three short statements:

```
A = B + C;  D = E + F;  Count = Count + 1
```

### White space

**Description**

Blanks, tabs, form feeds, and comments are forms of white space. The compiler treats white space as a delimiter and does not consider the number of white space characters.

**Usage**

**White space in string literals**  The number of white space characters is preserved when they are part of a string literal (enclosed in single or double quotation marks).
Dashes in identifiers  Unless you have prohibited the use of dashes in identifiers (see “Identifier names” on page 5), you must surround a dash used as a minus sign with spaces. Otherwise, PowerBuilder considers the dash as part of a variable name:

\[
\text{Order - Balance} \quad \text{// Subtracts Balance from Order} \\
\text{Order-Balance} \quad \text{// A variable named Order-Balance}
\]

Examples

**Example 1**  Here the spaces and the comment are white space, so the compiler ignores them:

\[
A + B \quad /*\text{Adjustment factor} */ + C
\]

**Example 2**  Here the spaces are within a string literal, so the compiler does not ignore them:

\[
"\text{The value of A + B is:}"\]

Conditional compilation

**Description**  The use of conditional compilation directives causes the PowerBuilder preprocessor to parse blocks of code before they are passed to the compiler.

**Syntax**  

\[
\#IF \{ \text{NOT } \} \text{DEFINED} \quad \text{predefined\_symbols} \quad \text{THEN} \\
\quad \text{action1} \\
\quad \{ \#ELSEIF \text{DEFINED} \quad \text{predefined\_symbols} \quad \text{THEN} \\
\quad \quad \text{action2} \} \\
\quad \{ \#ELSE \quad \text{action3} \} \\
\#END IF
\]

**Parameter** | **Description**
--- | ---
**predefined\_symbols** | A predefined identifier or a combination of predefined identifiers separated by AND or OR operators. In the current release, you cannot use a user-defined identifier.

**action1, action2, action3** | The action you want performed if the condition in the previous statement was met.

**Usage**  Conditional compilation enables you to include PowerScript code for a specific target type or set of target types in an application. You can also include debug code in your application and specify in the Project painter whether it will be included in your application’s executable file.
Conditional compilation

The preprocessor substitutes blank lines for statements with a leading number (#) sign character. It passes the code in the action statements to the compiler or converts it to blank lines depending on whether the condition in the previous preprocessor directive was met.

The following table displays the predefined symbols, the project types to which they correspond, and their effects on the code passed to the compiler.

**Table 1-5: Predefined symbols for conditional compilation**

<table>
<thead>
<tr>
<th>Predefined symbols</th>
<th>Target type</th>
<th>Code in this processing block</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBNATIVE</td>
<td>Standard PowerBuilder client-server or distributed applications</td>
<td>Fully parsed for the standard application and converted to blank lines for .NET targets.</td>
</tr>
<tr>
<td>PBWEBFORM</td>
<td>.NET Web Forms applications</td>
<td>Fully parsed for .NET Web Forms targets and converted to blank lines for all other targets.</td>
</tr>
<tr>
<td>PBWINFORM</td>
<td>.NET Windows Forms applications</td>
<td>Fully parsed for .NET Windows Forms targets and converted to blank lines for all other targets.</td>
</tr>
<tr>
<td>PBWEBSERVICE</td>
<td>.NET Web Service component targets</td>
<td>Fully parsed for .NET Web Service targets and converted to blank lines for all other targets.</td>
</tr>
<tr>
<td>PBDOTNET</td>
<td>.NET Web Forms and Windows Forms applications, and .NET Assembly and .NET Web Service components</td>
<td>Fully parsed for all .NET targets and converted to blank lines for all other targets.</td>
</tr>
<tr>
<td>DEBUG</td>
<td>All PowerBuilder standard and .NET targets</td>
<td>When a project’s Enable DEBUG Symbol check box is selected, code is fully parsed by the compiler and included in the deployed application. The code is converted to blank lines when the check box is cleared. The DEBUG symbol is always defined in the development environment.</td>
</tr>
</tbody>
</table>
You can use the NOT operator to include code for all target types that are not of the type that you specify, and you can use AND and OR operators to combine symbols. For example, code that follows this statement will be parsed for all targets except standard PowerBuilder applications and .NET Windows Forms targets:

```plaintext
#if NOT defined PBNATIVE OR PBWINFORM then
```

Comments can be added to conditional code blocks if they are preceded by double slash marks (`//`) in the same line of code. You cannot use the PowerScript line continuation character (`&`) in a conditional code statement. You must use it in code that you embed in the conditional block when you use more than one line for a single line of code.

Conditional compilation is not supported in DataWindow syntax, or in structure or menu objects. You cannot edit the source code for an object to include conditional compilation blocks that span function, event, or variable definition boundaries.

You must rebuild your application after you add a DEBUG conditional block.

The following table shows the types of error messages displayed for incorrect conditional compilation code.

<table>
<thead>
<tr>
<th>Error message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid if statement</td>
<td>#if statement without a defined symbol, with an incorrectly defined symbol, or without a <code>then</code> clause</td>
</tr>
<tr>
<td>#end if directive expected</td>
<td>#if statement without an <code>#end if</code> statement</td>
</tr>
<tr>
<td>Unexpected preprocessor directive</td>
<td>Caused by an <code>#else</code>, <code>#elseif</code>, or <code>#end if</code> statement when not preceded by an <code>#if</code> statement</td>
</tr>
<tr>
<td>Preprocessor syntax error</td>
<td>Caused by including text after an <code>#else</code> or <code>#end if</code> statement when the text is not preceded by comment characters (//)</td>
</tr>
</tbody>
</table>

Examples

When you run or debug the application in the development environment, the following code is always parsed and you always see the message box. When you run the executable file, the code is parsed only if the DEBUG symbol is enabled on the General page in the Project painter:

```plaintext
#if defined DEBUG then
    MessageBox("Debugging","Ctr value is " + string(i))
#end if
```
Conditional compilation

For examples of using conditional compilation in .NET targets, see the section on conditional compilation in *Deploying Applications and Components to .NET*. 
CHAPTER 2  

Datatypes

About this chapter
This chapter describes the PowerScript datatypes.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard datatypes</td>
<td>21</td>
</tr>
<tr>
<td>The Any datatype</td>
<td>26</td>
</tr>
<tr>
<td>System object datatypes</td>
<td>29</td>
</tr>
<tr>
<td>Enumerated datatypes</td>
<td>30</td>
</tr>
<tr>
<td>PowerBuilder datatypes in EAServer</td>
<td>31</td>
</tr>
</tbody>
</table>

Standard datatypes

The datatypes
The standard datatypes in PowerBuilder are the familiar datatypes that are used in many programming languages, including char, integer, decimal, long, and string. In PowerScript, you use these datatypes to declare variables or arrays.

These are the standard PowerScript datatypes, followed by a description of each:

- Blob
  - Integer or Int
- Boolean
  - LongLong
- Byte
  - Long
- Char or character
  - Real
- Date
  - String
- DateTime
  - Time
- Decimal or Dec
  - UnsignedInteger, UnsignedInt, or UInt
- Double
  - UnsignedLong or ULong

Blob
Binary large object. Used to store an unbounded amount of data (for example, generic binary, image, or large text such as a word-processing document).

Boolean
Contains true or false.
Standard datatypes

Byte
8-bit unsigned integers, from 0 to +255.

Using literals To assign a literal value, use any whole positive number in the range 0 to 255. The leading plus sign is not required (18 and +18 are the same). For example:

```
1 123 200 +55 +200
```

Char or character
A single ASCII character.

If you have character-based data that you will want to parse in an application, you might want to define it as an array of type char. Parsing a char array is easier and faster than parsing strings. If you will be passing character-based data to external functions, you might want to use char arrays instead of strings.

For more information about passing character-based data to external functions, see Application Techniques. For information about datatype conversion when assigning strings to chars and vice versa, see “String and char datatypes in PowerBuilder” on page 78.

Using literals To assign a literal value, enclose the character in either single or double quotation marks. For example:

```
char c
  c = 'T'
  c = "T"
```

Date
The date, including the full year (1000 to 3000), the number of the month (01 to 12), and the day (01 to 31).

Using literals To assign a literal value, separate the year, month, and day with hyphens. For example:

```
2001-12-25  // December 25, 2001
2003-02-06  // February 6, 2003
```

DateTime
The date and time in a single datatype, used only for reading and writing DateTime values from and to a database. To convert DateTime values to datatypes that you can use in PowerBuilder, use:

- The Date(datetime) function to convert a DateTime value to a PowerBuilder date value after reading from a database
- The Time(datetime) function to convert a DateTime value to a PowerBuilder time value after reading from a database
- The DateTime(date, time) function to convert a date and (optional) time to a DateTime before writing to a DateTime column in a database.

PowerBuilder supports microseconds in the database interface for any DBMS that supports microseconds.
Decimal or Dec

Signed decimal numbers, positive or negative, with up to 28 digits. You can place the decimal point anywhere within the 28 digits—for example, 123.456, 0.000000000000000000000001 or 12345678901234.5678901234.

Using literals

To assign a literal value, use any number with a decimal point and no exponent. The plus sign is optional (95 and +95 are the same). For numbers between zero and one, the zero to the left of the decimal point is optional (for example, 0.1 and .1 are the same). For whole numbers, zeros to the right of the decimal point are optional (32.00, 32.0, and 32. are all the same). For example:

```
12.34  0.005 14.0 -6500  +3.5555
```

Double

A signed floating-point number with 15 digits of precision and a range from 2.2250738585073E-308 to 1.79769313486231E+308, and -2.2250738585073E-308 to -1.79769313486231E+308.

Using literals

Use literals as for integers, but longer numbers are permitted.

Integer or Int

16-bit signed integers, from -32768 to +32767.

Using literals

To assign a literal value, use any whole number (positive, negative, or zero). The leading plus sign is optional (18 and +18 are the same). For example:

```
1 123 1200 +55 -32
```

Long

32-bit signed integers, from -2147483648 to +2147483647.

Using literals

Use literals as for integers, but longer numbers are permitted.

LongLong

64-bit signed integers, from -9223372036854775808 to 9223372036854775807.

Using literals

Use literals as for integers, but longer numbers are permitted.

Real

A signed floating-point number with six digits of precision and a range from 3.402822E-38 to 3.402822E+38, and -3.402822E-38 to -3.402822E+38.

Using literals

To assign a literal value, use a decimal value, followed by E, followed by an integer; no spaces are allowed. The decimal number before the E follows all the conventions specified above for decimal literals. The leading plus sign in the exponent (the integer following the E) is optional (3E5 and 3E+5 are the same). For example:

```
2E4   2.5E38  +6.02E3  -4.1E-2
-7.45E16   7.7E+8   3.2E-38
```

String

Any ASCII character with variable length (0 to 2147483647).
Most of the character-based data in your application, such as names, addresses, and so on, will be defined as strings. PowerScript provides many functions that you can use to manipulate strings, such as a function to convert characters in a string to uppercase and functions to remove leading and trailing blanks.

For more information about passing character-based data to external functions, see Application Techniques. For information about datatype conversion when assigning strings to chars and vice versa, see “String and char datatypes in PowerBuilder” on page 78.

Using literals  To assign a literal value, enclose as many as 1024 characters in either single or double quotes, including a string of zero length or an empty string. For example:

```powerbuilder
string sl
sl = 'This is a string'
sl = "This is a string"
```

You can embed a quotation mark in a string literal if you enclose the literal with the other quotation mark. For example, the following statements result in the string Here's a string:

```powerbuilder
string sl
sl = "Here's a string."
```

You can also use a tilde (~) to embed a quotation mark in a string literal. For example:

```powerbuilder
string sl = 'He said, "It~'s good!"
```

Complex nesting  When you nest a string within a string that is nested in another string, you can use tildes to tell the parser how to interpret the quotation marks. Each pass through the parser strips away the outermost quotes and interprets the character after each tilde as a literal. Two tildes become one tilde, and tilde-quote becomes the quote alone.

**Example 1**  This string has two levels of nesting:

```
"He said ~"she said ~"Hi ~" ~" 
```

The first pass results in:

```
He said "she said ~"Hi ~"
```

The second pass results in:

```
she said ~"Hi"
```

The third pass results in:

```
Hi
```
Example 2  A more probable example is a string for the Modify function that sets a DataWindow® property. The argument string often requires complex quotation marks (because you must specify one or more levels of nested strings). To understand the quotation marks, consider how PowerBuilder will parse the string. The following string is a possible argument for the Modify function; it mixes single and double quotes to reduce the number of tildes:

"bitmap_1.Invert='0-tIf(empstatus='A--',0,1)"

The double quotes tell PowerBuilder to interpret the argument as a string. It contains the expression being assigned to the Invert property, which is also a string, so it must be quoted. The expression itself includes a nested string, the quoted A. First, PowerBuilder evaluates the argument for Modify and assigns the single-quoted string to the Invert property. In this pass through the string, it converts two tildes to one. The string assigned to Invert becomes:

'0[tab]If(empstatus='A--',0,1)'

Finally, PowerBuilder evaluates the property’s expression, converting tilde-quote to quote, and sets the bitmap’s colors accordingly.

Example 3  There are many ways to specify quotation marks for a particular set of nested strings. The following expressions for the Modify function all have the same end result:

"emp.Color = ~"0~tIf(stat="a---",255,16711680)~"
"emp.Color = ~"0~tIf(stat='a--',255,16711680)~"
"emp.Color = '0~tIf(stat='a--',255,16711680)'
"emp.Color = ~"0~tIf(stat='a',255,16711680)~"

Rules for quotation marks and tildes  When nesting quoted strings, the following rules of thumb might help:

- A tilde tells the parser that the next character should be taken as a literal, not a string terminator
- Pairs of single quotes (‘’) can be used in place of pairs of tilde double quotes (~”)
- Pairs of tilde tilde single quotes (~~’) can be used in place of pairs of triple tilde double quotes (~~~”)

Time  The time in 24-hour format, including the hour (00 to 23), minute (00 to 59), second (00 to 59), and fraction of second (up to six digits), with a range from 00:00:00 to 23:59:59:999999.

PowerBuilder supports microseconds in the database interface for any DBMS that supports microseconds.
**The Any datatype**

**Using literals**  The time in 24-hour format, including the hour (00 to 23), minute (00 to 59), second (00 to 59), and fraction of second (up to six digits), with a range from 00:00:00 to 23:59:59.999999. You separate parts of the time with colons—except for fractional sections, which should be separated by a decimal point. For example:

- 21:09:15 // 15 seconds after 9:09 pm
- 06:00:00 // Exactly 6 am
- 10:29:59 // 1 second before 10:30 am
- 10:29:59.9 // 1/10 sec before 10:30 am

---

**UnsignedInteger, UnsignedInt, or UInt**  16-bit unsigned integers, from 0 to 65535.

**UnsignedLong or ULong**  32-bit unsigned integers, from 0 to 4294967295.

---

**The Any datatype**

**General information**  PowerBuilder also supports the Any datatype, which can hold any kind of value, including standard datatypes, objects, structures, and arrays. A variable whose type is Any is a chameleon datatype—it takes the datatype of the value assigned to it.

---

**Do not use Any in EAServer component definition**  The Any datatype is specific to PowerScript and is not supported in the IDL of an EAServer component. CORBA has a datatype called Any that can assume any legal IDL type at runtime, but it is not semantically equivalent to the PowerBuilder Any type. You must exclude the PowerBuilder Any datatype from the component interface definition, but you can use it within the component.

---

**Declarations and assignments**  You declare Any variables just as you do any other variable. You can also declare an array of Any variables, where each element of the array can have a different datatype.

You assign data to Any variables with standard assignment statements. You can assign an array to a simple Any variable.
After you assign a value to an Any variable, you can test the variable with the ClassName function and find out the actual datatype:

```plaintext
any la_spreadsheetdata
la_spreadsheetdata = ole_1.Object.cells(1,1).value
CHOOSE CASE ClassName(la_spreadsheetdata)
   CASE "integer"
      ...
   CASE "string"
      ...
END CHOOSE
```

These rules apply to Any assignments:

- You can assign anything into an Any variable.
- You must know the content of an Any variable to make assignments from the Any variable to a compatible datatype.

**Restrictions**

If the value of a simple Any variable is an array, you cannot access the elements of the array until you assign the value to an array variable of the appropriate datatype. This restriction does not apply to the opposite case of an array of Any variables—you can access each Any variable in the array.

If the value of an Any variable is a structure, you cannot use dot notation to access the elements of the structure until you assign the value to a structure of the appropriate datatype.

After a value has been assigned to an Any variable, it cannot be converted back to a generic Any variable without a datatype. Even if you set it to NULL, it retains the datatype of the assigned value until you assign another value.

**Operations and expressions**

You can perform operations on Any variables as long as the datatype of the data in the Any variable is appropriate to the operator. If the datatype is not appropriate to the operator, an execution error occurs.

For example, if instance variables `ia_1` and `ia_2` contain numeric data, this statement is valid:

```plaintext
any la_3
la_3 = ia_1 - ia_2
```

If `ia_1` and `ia_2` contain strings, you can use the concatenation operator:

```plaintext
any la_3
la_3 = ia_1 + ia_2
```

However, if `ia_1` contained a number and `ia_2` contained a string, you would get an execution error.
**The Any datatype**

**Datatype conversion functions**  PowerScript datatype conversion functions accept Any variables as arguments. When you call the function, the Any variable must contain data that can be converted to the specified type.

For example, if `ia_any` contains a string, you can assign it to a string variable:

```powerbuilder
ls_string = ia_any
```

If `ia_any` contains a number that you want to convert to a string, you can call the `String` function:

```powerbuilder
ls_string = String(ia_any)
```

**Other functions**  If a function’s prototype does not allow Any as a datatype for an argument, you cannot use an Any variable without a conversion function, even if it contains a value of the correct datatype. When you compile the script, you get compiler errors such as Unknown function or Function not found.

For example, the argument for the `Len` function refers to a string column in a DataWindow, but the expression itself has a type of Any:

```powerbuilder
IF Len(dw_notes.Object.Notes[1]) > 0 THEN  // Invalid
```

This works because the string value of the Any expression is explicitly converted to a string:

```powerbuilder
IF Len(String(dw_notes.Object.Notes[1])) > 0 THEN
```

**Expressions whose datatype is Any**  Expressions that access data whose type is unknown when the script is compiled have a datatype of Any. These expressions include expressions or functions that access data in an OLE object or a DataWindow object:

```powerbuilder
myoleobject.application.cells(1,1).value
dw_1.Object.Data[1,1]
dw_1.Object.Data.empid[99]
```

The objects these expressions point to can change so that the type of data being accessed also changes.

Expressions that refer to DataWindow data can return arrays and structures and arrays of structures as Any variables. For best performance, assign the DataWindow expression to the appropriate array or structure without using an intermediate Any variable.
Overusing the Any datatype

Do not use Any variables as a substitute for selecting the correct datatype in your scripts. There are two reasons for this:

- **At execution time, using Any variables is slow**  
  PowerBuilder must do much more processing to determine datatypes before it can make an assignment or perform an operation involving Any variables. In particular, an operation performed many times in a loop will suffer greatly if you use Any variables instead of variables of the appropriate type.

- **At compile time, using Any variables removes a layer of error checking from your programming**  
  The PowerBuilder compiler makes sure datatypes are correct before code gets executed. With Any variables, some of the errors that can be caught by the compiler are not found until the code is run.

System object datatypes

Objects as datatypes

System object datatypes are specific to PowerScript. You view a list of all the system objects by selecting the System tab in the Browser.

In building PowerBuilder applications, you manipulate objects such as windows, menus, CommandButtons, ListBoxes, and graphs. Internally, PowerBuilder defines each of these kinds of objects as a datatype. Usually you do not need to concern yourself with these objects as datatypes—you simply define the objects in a PowerBuilder painter and use them.

However, sometimes you need to understand how PowerBuilder maintains its system objects in a hierarchy of datatypes. For example, when you need to define instances of a window, you define variables whose datatype is window. When you need to create an instance of a menu to pop up in a window, you define a variable whose datatype is menu.

PowerBuilder maintains its system objects in a class hierarchy. Each type of object is a class. The classes form an inheritance hierarchy of ancestors and descendants.

Examples

All the classes shown in the Browser are actually datatypes that you can use in your applications. You can define variables whose type is any class.

For example, the following code defines window and menu variables:

```plaintext
window mywin
menu mymenu
```
If you have a series of buttons in a window and need to keep track of one of them (such as the last one clicked), you can declare a variable of type CommandButton and assign it the appropriate button in the window:

```
// Instance variable in a window
commandbutton LastClicked
// In Clicked event for a button in the window.
// Indicates that the button was the last one clicked by the user.
LastClicked = This
```

Because it is a CommandButton, the LastClicked variable has all the properties of a CommandButton. After the last assignment above, LastClicked’s properties have the same values as the most recently clicked button in the window.

To learn more about working with instances of objects through datatypes, see “About objects” on page 80.

### Enumerated datatypes

Like the system object datatypes, enumerated datatypes are specific to PowerScript. Enumerated datatypes are used in two ways:

- As arguments in functions
- To specify the properties of an object or control

You can list all the enumerated datatypes and their values by selecting the Enumerated tab in the Browser.

You cannot create your own enumerated datatypes. As an alternative, you can declare a set of constant variables and assign them initial values. See “Declaring constants” on page 47.

A variable of one of the enumerated datatypes can be assigned a fixed set of values. Values of enumerated datatypes always end with an exclamation point (!). For example, the enumerated datatype Alignment, which specifies the alignment of text, can be assigned one of the following three values: Center!, Left!, and Right!:

```
mle_edit.Alignment=Right!
```
Incorrect syntax
Do not enclose an enumerated datatype value in quotation marks. If you do, you receive a compiler error.

Enumerated datatypes have an advantage over standard datatypes. When an enumerated datatype is required, the compiler checks the data and makes sure it is the correct type. For example, if you set an enumerated datatype variable to any other datatype or to an incorrect value, the compiler does not allow it.

PowerBuilder datatypes in EAServer

Prior to EAServer 6.0, all EAServer component interfaces were defined in standard CORBA IDL. The following table lists the predefined datatypes used in EAServer Manager, the equivalent CORBA IDL types, and the PowerBuilder datatypes that they map to.

For information about datatype mapping in EAServer 6.0, see the PowerBuilder components chapter in the CORBA Components Guide on the Sybase Product Manuals Web site.

Table 2-1: PowerBuilder datatypes in EAServer

<table>
<thead>
<tr>
<th>EAServer Manager</th>
<th>CORBA IDL</th>
<th>PowerBuilder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer (16-bit)</td>
<td>Short</td>
<td>Integer</td>
</tr>
<tr>
<td>Integer (32-bit)</td>
<td>Long</td>
<td>Long</td>
</tr>
<tr>
<td>Integer (64-bit)</td>
<td>Long long</td>
<td>LongLong</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>Float</td>
<td>Float</td>
<td>Real</td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>Binary</td>
<td>BCD::Binary</td>
<td>Blob</td>
</tr>
<tr>
<td>Decimal</td>
<td>BCD::Decimal</td>
<td>Decimal</td>
</tr>
<tr>
<td>Money</td>
<td>BCD::Money</td>
<td>Decimal</td>
</tr>
<tr>
<td>Date</td>
<td>MJD::Date</td>
<td>Date</td>
</tr>
<tr>
<td>Time</td>
<td>MJD::Time</td>
<td>Time</td>
</tr>
<tr>
<td>Timestamp</td>
<td>MJD::Timestamp</td>
<td>DateTime</td>
</tr>
<tr>
<td>ResultSet</td>
<td>TabularResults::ResultSet</td>
<td>ResultSet</td>
</tr>
<tr>
<td>ResultSets</td>
<td>TabularResults::ResultSets</td>
<td>ResultSets</td>
</tr>
<tr>
<td>Void</td>
<td>Void</td>
<td>None</td>
</tr>
</tbody>
</table>
CHAPTER 3  Declarations

About this chapter
This chapter explains how to declare variables, constants, and arrays and refer to them in scripts, and how to declare remote procedure calls (RPCs) and external functions that reside in dynamic link libraries (DLLs).

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaring variables</td>
<td>33</td>
</tr>
<tr>
<td>Declaring constants</td>
<td>47</td>
</tr>
<tr>
<td>Declaring arrays</td>
<td>48</td>
</tr>
<tr>
<td>Declaring external functions</td>
<td>57</td>
</tr>
<tr>
<td>Declaring DBMS stored procedures as remote procedure calls</td>
<td>65</td>
</tr>
</tbody>
</table>

Declaring variables

General information
Before you use a variable in a PowerBuilder script, you must declare it (give it a datatype and a name).

A variable can be a standard datatype, a structure, or an object. Object datatypes can be system objects as displayed in the Browser or they can be objects you have defined by deriving them from those system object types. For most variables, you can assign it a value when you declare it. You can always assign it a value within a script.
Declaring variables

Where to declare variables

You determine the scope of a PowerScript variable by selecting where you declare it. Instance variables have additional access keywords that restrict specific scripts from accessing the variable.

The following table shows the four scopes of variables.

Table 3-1: PowerScript variable scopes

<table>
<thead>
<tr>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Accessible anywhere in the application. It is independent of any object definition.</td>
</tr>
<tr>
<td>Instance</td>
<td>Belongs to an object and is associated with an instance of that object (you can think of it as a property of the object). Instance variables have access keywords that determine whether scripts of other objects can access them. They can belong to the application object, a window, a user object, or a menu.</td>
</tr>
<tr>
<td>Shared</td>
<td>Belongs to an object definition and exists across all instances of the object. Shared variables retain their value when an object is closed and opened again. Shared variables are always private. They are accessible only in scripts for the object and for controls associated with the object. They can belong to the application object, a window, a user object, or a menu.</td>
</tr>
<tr>
<td>Local</td>
<td>A temporary variable that is accessible only in the script in which you define it. When the script has finished executing, the variable constant ceases to exist.</td>
</tr>
</tbody>
</table>

Global, instance, and shared declarations

Global, instance, and shared variables can be defined in the Script view of the Application, Window, User Object, or Menu painters. Global variables can also be defined in the Function painter:

1. Select Declare from the first drop-down list in the Script view.
2. Select the type of variable you want to declare in the second drop-down list of the Script view.
3. Type the declaration in the scripting area of the Script view.

Local declarations

You declare local variables for an object or control in the script for that object or control.

Declaring SQL cursors

You can also declare SQL cursors that are global, shared, instance, or local. Open a specific script or select a variable declaration scope in the Script view and type the DECLARE SQL statement or select Paste SQL from the PainterBar or pop-up menu.
About using variables

To use or set a variable's value in a PowerBuilder script, you name the variable. The variable must be known to the compiler—in other words, it must be in scope.

You can use a variable anywhere you need its value—for example, as a function argument or in an assignment statement.

How PowerBuilder looks for variables

When PowerBuilder executes a script and finds an unqualified reference to a variable, it searches for the variable in the following order:

1. A local variable
2. A shared variable
3. A global variable
4. An instance variable

As soon as PowerBuilder finds a variable with the specified name, it uses the variable’s value.

Referring to global variables

To refer to a global variable, you specify its name in a script. However, if the global variable has the same name as a local or shared variable, the local or shared variable will be found first.

To refer to a global variable that is masked by a local or shared variable of the same name, use the global scope operator (::) before the name:

```
::globalname
```

For example, this statement compares the value of local and global variables, both named total:

```
IF total < ::total THEN ...
```

Referring to instance variables

You can refer to an instance variable in a script if there is an instance of the object open in the application. Depending on the situation, you might need to qualify the name of the instance variable with the name of the object defining it.

Using unqualified names

You can refer to instance variables without qualifying them with the object name in the following cases:

- For application-level variables, in scripts for the application object
- For window-level variables, in scripts for the window itself and in scripts for controls in that window
Declaring variables

- For user-object-level variables, in scripts for the user object itself and in scripts for controls in that user object.
- For menu-level variables, in scripts for a menu object, either the highest-level menu or scripts for the menu objects included as items on the menu.

For example, if `w_emp` has an instance variable `EmpID`, then you can reference `EmpID` without qualification in any script for `w_emp` or its controls as follows:

```powerbuilder
sle_id.Text = EmpID
```

**Using qualified names**

In all other cases, you need to qualify the name of the instance variable with the name of the object using dot notation:

```powerbuilder
object.instancevariable
```

This requirement applies only to Public instance variables. You cannot reference Private instance variables outside the object at all, qualified or not.

For example, to refer to the `w_emp` instance variable `EmpID` from a script outside the window, you need to qualify the variable with the window name:

```powerbuilder
sle_ID.Text = w_emp.EmpID
```

There is another situation in which references must be qualified. Suppose that `w_emp` has an instance variable `EmpID` and that in `w_emp` there is a `CommandButton` that declares a local variable `EmpID` in its Clicked script. In that script, you must qualify all references to the instance variable:

```powerbuilder
Parent.EmpID
```

**Using pronouns as name qualifiers**

To avoid ambiguity when referring to variables, you might decide to always use qualified names for object variables. Qualified names leave no doubt about whether a variable is local, instance, or shared.

To write generic code but still use qualified names, you can use the pronouns `This` and `Parent` to refer to objects. Pronouns keep a script general by allowing you to refer to the object without naming it specifically.

**Window variables in window scripts**

In a window script, use the pronoun `This` to qualify the name of a window instance variable. For example, if a window has an instance variable called `index`, then the following statements are equivalent in a script for that window, as long as there is no local or global variable named `index`:

```powerbuilder
index = 5
This.index = 5
```
**Window variables in control scripts**  In a script for a control in a window, use the pronoun Parent to qualify the name of a window instance variable—the window is the parent of the control. In this example, the two statements are equivalent in a script for a control in that window, as long as there is no local or global variable named “index”:

```
index = 5
Parent.index = 5
```

**Naming errors**  If a local or global exists with the name “index,” then the unqualified name refers to the local or global variable. It is a programming error if you meant to refer to the object variable. You get an informational message from the compiler if you use the same name for instance and global variables.

---

**Syntax of a variable declaration**

**Simple syntax**  In its simplest form, a PowerScript variable declaration requires only two parts: the datatype and the variable name. For example:

```
datatype variablename
```

**Full syntax**  The full syntax allows you to specify access and an initial value. Arrays and some datatypes, such as blobs and decimals, accept additional information:

```
{ access } datatype { { size } } { { precision } } variablename { = value }
{ variablename2 { = value2 } }
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>access</code> (optional)</td>
<td>(For instance variables only) Keywords specifying the access for the variable. For information, see “Access for instance variables” on page 42.</td>
</tr>
<tr>
<td><code>datatype</code></td>
<td>The datatype of the variable. You can specify a standard datatype, a system object, or a previously defined structure. For blobs and decimals, you can specify the size or precision of the data by including an optional value in brackets.</td>
</tr>
<tr>
<td><code>{ size }</code> (optional)</td>
<td>(For blobs only) A number, enclosed in braces, specifying the size in bytes of the blob. If <code>{ size }</code> is omitted, the blob has an initial size of zero and PowerBuilder adjusts its size each time it is used at runtime. If you enter a size that exceeds the declared length in a script, PowerBuilder truncates the blob data.</td>
</tr>
</tbody>
</table>
Declaring variables

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ precision / (optional)</td>
<td>(For decimals only) A number, enclosed in braces, specifying the number of digits after the decimal point. If you do not specify a precision, the variable takes the precision assigned to it in the script.</td>
</tr>
<tr>
<td>variablename</td>
<td>The name of the variable (must be a valid PowerScript identifier, as described in “Identifier names” on page 5). You can define additional variables with the same datatype by naming additional variable names, separated by commas; each variable can have a value.</td>
</tr>
<tr>
<td>value (optional)</td>
<td>A literal or expression of the appropriate datatype that will be the initial value of the variable. Blobs cannot be initialized with a value. For information, see “Initial values for variables” on page 40.</td>
</tr>
</tbody>
</table>

Examples

Declaring instance variables

```powerbuilder
integer ii_total = 100 // Total shares
date id_date // Date shares were bought
```

Declaring a global variable

```powerbuilder
string gs_name
```

Declaring shared variables

```powerbuilder
time st_process_start
string ss_process_name
```

Declaring local variables

```powerbuilder
string ls_city = "Boston"
integer li_count
```

Declaring blobs

This statement declares `ib_Emp_Picture` a blob with an initial length of zero. The length is adjusted when data is assigned to it:

```powerbuilder
blob ib_Emp_Picture
```

This statement declares `ib_Emp_Picture` a blob with a fixed length of 100 bytes:

```powerbuilder
blob{100} ib_Emp_Picture
```

Declaring decimals

These statements declare shared variables `sc_Amount` and `sc_dollars_accumulated` as decimal numbers with two digits after the decimal point:

```powerbuilder
decimal{2} sc_Amount
decimal{2} sc_dollars_accumulated
```
This statement declares `lc_Rate1` and `lc_Rate2` as decimal numbers with four digits after the decimal point:

```
dec{4} lc_Rate1, lc_Rate2
```

This statement declares `lc_Balance` as a decimal with zero digits after the decimal point:

```
decimal{0} lc_Balance
```

This statement does not specify the number of decimal places for `lc_Result`. After the product of `lc_Op1` and `lc_Op2` is assigned to it, `lc_Result` has four decimal places:

```
dec lc_Result
dec{2} lc_Op1, lc_Op2
lc_Result = lc_Op1 * lc_Op2
```

**Datatype of a variable**

A PowerScript variable can be declared as one of the following datatypes:

- A standard datatype (such as an integer or string).
- An object or control (such as a window or CommandButton).
- An object or structure that you have defined (such as a window called `mywindow`). An object you have defined must be in a library on the application’s library search path when the script is compiled.

**Variable names**

In a well-planned application, standards determine how you name your PowerScript variables. Naming conventions make scripts easy to understand and help you avoid name conflicts. A typical approach is to include a prefix that identifies the scope and the datatype of the variable. For example, a prefix for an instance variable’s name typically begins with `i` (such as `ii_count` or `is_empname`), a local integer variable’s name would be `li_total` and a global integer variable’s name would be `gi_total`.

For information about naming conventions, see the PowerBuilder *Users Guide*. 
Declaring variables

X and Y as variable names

Although you might think of x and y as typical variable names, in PowerBuilder they are also properties that specify an object’s onscreen coordinates. If you use them as variables and forget to declare them, you do not get a compiler error. Instead, PowerBuilder assumes you want to move the object, which might lead to unexpected results in your application.

Initial values for variables

When you declare a PowerScript variable, you can accept the default initial value or specify an initial value in the declaration.

If you do not initialize a variable when you declare it, PowerBuilder sets the variable to the default value for its datatype as shown in the following table.

Table 3-3: Default initial values for variables

<table>
<thead>
<tr>
<th>For this variable datatype</th>
<th>PowerBuilder sets this default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob</td>
<td>A blob of 0 length; an empty blob</td>
</tr>
<tr>
<td>Char (or character)</td>
<td>ASCII value 0</td>
</tr>
<tr>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>Date</td>
<td>1900-01-01 (January 1, 1900)</td>
</tr>
<tr>
<td>DateTime</td>
<td>1900-01-01 00:00:00</td>
</tr>
<tr>
<td>Numeric (byte, integer, long, longlong, decimal, real, double, UnsignedInteger, and UnsignedLong)</td>
<td>0</td>
</tr>
<tr>
<td>String</td>
<td>Empty string (&quot;&quot;)</td>
</tr>
<tr>
<td>Time</td>
<td>00:00:00 (midnight)</td>
</tr>
</tbody>
</table>

Specifying a literal as an initial value

To initialize a variable when you declare it, place an equal sign (=) and a literal appropriate for that variable datatype after the variable. For information about literals for specific datatypes, see “Standard datatypes” on page 21.

Do not use a function’s return value

You should not initialize a variable by assigning it the return value of a global user defined function, because it might not compile correctly, or because it could lead to confusion about the value assigned. For example, do not use:

```powerbuilder
integer i = f_return_one()
```

Although you can use global system functions or expressions to initialize variables with compile time values in a variable declaration statement, for runtime value assignments, you must also declare variables and assign their values in separate statements.
This example declares `li_count` as an integer whose value is 5:

```powershell
integer li_count=5
```

This example declares `li_a` and `li_b` as integers and initializes `li_a` to 5 and `li_b` to 10:

```powershell
integer li_a=5, li_b=10
```

This example initializes `ls_method` with the string "UPS":

```powershell
string ls_method="UPS"
```

This example initializes `ls_headers` to three words separated by tabs:

```powershell
string ls_headers = "Name~tAddress~tCity"
```

This example initializes `li_a` to 1 and `li_c` to 100, leaving `li_b` set to its default value of zero:

```powershell
integer li_a=1, li_b, li_c=100
```

This example declares `ld_StartDate` as a date and initializes it with the date February 1, 2004:

```powershell
date ld_StartDate = 2004-02-01
```

Specifying an expression as an initial value

You can initialize a variable with the value of an existing variable or expression, such as:

```powershell
integer i = 100
integer j = i
```

When you do this, the second variable is initialized with the value of the expression when the script is compiled. The initialization is not reevaluated at runtime.

**If the expression’s value changes**. Because the expression’s value is set to the variable when the script is compiled (not at runtime) make sure the expression is not one whose value is based on current conditions. If you want to specify an expression whose value will be different when the application is executed, do not initialize the variable in the declaration. For such values, declare the variable and assign the value in separate statements.

In this declaration, the value of `d_date` is the date the script is compiled:

```powershell
date d_date = Today( )
```

In contrast, these statements result in `d_date` being set to the date the application is run:

```powershell
date d_date
d_date = Today( )
```
### Declaring variables

#### How shared variables are initialized

When you use a shared variable in a script, the variable is initialized when the first instance of the object is opened. When the object is closed, the shared variable continues to exist until you exit the application. If you open the object again without exiting the application, the shared variable will have the value it had when you closed the object.

For example, if you set the shared variable `Count` to 20 in the script for a window, then close the window, and then reopen the window without exiting the application, `Count` will be equal to 20.

#### When using multiple instances of windows

If you have multiple instances of the window in the example above, `Count` will be equal to 20 in each instance. Since shared variables are shared among all instances of the window, changing `Count` in any instance of the window changes it for all instances.

#### How instance variables are initialized

When you define an instance variable for a window, menu, or application object, the instance variable is initialized when the object is opened. Its initial value is the default value for its datatype or the value specified in the variable declarations.

When you close the object, the instance variable ceases to exist. If you open the object again, the instance variable is initialized again.

#### When to use multiple instances of windows

When you build a script for one of multiple instances of a window, instance variables can have a different value in each instance of the window. For example, to set a flag based on the contents of the instance of a window, you would use an instance variable.

#### When to use shared variables instead

Use a shared variable instead of an instance variable if you need a variable that:

- Keeps the same value over multiple instances of an object
- Continues to exist after the object is closed

### Access for instance variables

**Description**

The general syntax for declaring PowerScript variables (see “Syntax of a variable declaration” on page 37) showed that you can specify access keywords in a declaration for an instance variable. This section describes those keywords.
When you specify an access right for a variable, you are controlling the visibility of the variable or its visibility access. Access determines which scripts recognize the variable’s name.

For a specified access right, you can control operational access with modifier keywords. The modifiers specify which scripts can read the variable’s value and which scripts can change it.

Syntax

{ access-right } { readaccess } { writeaccess } datatype variablename

The following table describes the parameters you can use to specify access rights for instance variables.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>access-right</td>
<td>A keyword specifying where the variable’s name will be recognized. Values are:</td>
</tr>
<tr>
<td>(optional)</td>
<td>• PUBLIC – (Default) Any script in the application can refer to the variable. In another object’s script, you use dot notation to qualify the variable name and identify the object it belongs to.</td>
</tr>
<tr>
<td></td>
<td>• PROTECTED – Scripts for the object for which the variable is declared and its descendants can refer to the variable.</td>
</tr>
<tr>
<td></td>
<td>• PRIVATE – Scripts for the object for which the variable is declared can refer to the variable. You cannot refer to the variable in descendants of the object.</td>
</tr>
<tr>
<td>readaccess</td>
<td>A keyword restricting the ability of scripts to read the variable’s value. Values are:</td>
</tr>
<tr>
<td>(optional)</td>
<td>• PROTECTEDREAD – Only scripts for the object and its descendants can read the variable.</td>
</tr>
<tr>
<td></td>
<td>• PRIVATEREAD – Only scripts for the object can read the variable.</td>
</tr>
</tbody>
</table>

When access-right is PUBLIC, you can specify either keyword. When access-right is PROTECTED, you can specify only PRIVATEREAD. You cannot specify a modifier for PRIVATE access, because PRIVATE is already fully restricted.

If readaccess is omitted, any script can read the variable.
Declaring variables

Usage
Access modifiers give you more control over which objects have access to a particular object’s variables. A typical use is to declare a public variable but only allow the owner object to modify it:

```
public protectedwrite integer ii_count
```

You can also group declarations that have the same access by specifying the access-right keyword as a label (see "Another format for access-right keywords" next).

When you look at exported object syntax, you might see the access modifiers SYSTEMREAD and SYSTEMWRITE. Only PowerBuilder can access variables with these modifiers. You cannot refer to variables with these modifiers in your scripts and functions and you cannot use these modifiers in your own definitions.

Examples
To declare these variables, select Declare>Instance Variables in the appropriate painter.

These declarations use access keywords to control the scripts that have access to the variables:

```
private integer ii_a, ii_n
public integer ii_Subtotal
protected integer ii_WinCount
```
This protected variable can only be changed by scripts of the owner object; descendants of the owner can read it:

```plaintext
protected privatewrite string is_label
```

These declarations have public access (the default) but can only be changed by scripts in the object itself:

```plaintext
privatewrite real ir_accum, ir_current_data
```

This declaration defines an integer that only the owner objects can write or read but whose name is reserved at the public level:

```plaintext
public privatered privaterwrite integer ii_reserved
```

**Private variable not recognized outside its object**  Suppose you have defined a window `w_emp` with a private integer variable `ii_int`:

```plaintext
private integer ii_int
```

In a script you declare an instance of the window called `w_myemp`. If you refer to the private variable `ii_int`, you get a compiler warning that the variable is not defined (because the variable is private and is not recognized in scripts outside the window itself):

```plaintext
w_emp w_myemp
w_myemp.ii_int = 1  // Variable not defined
```

**Public variable with restricted access**  Suppose you have defined a window `w_emp` with a public integer variable `ii_int` with write access restricted to private:

```plaintext
public privatewrite integer ii_int
```

If you write the same script as above, the compiler warning will say that you cannot write to the variable (the name is recognized because it is public, but write access is not allowed):

```plaintext
w_emp w_myemp
w_myemp.ii_int = 1  // Cannot write to variable
```
**Declaring variables**

### Another format for access-right keywords

**Description**
You can also group declarations of PowerScript variables according to access by specifying the access-right keyword as a label. It appears on its own line, followed by a colon (:).

**Syntax**
```
access-right:
{ readaccess }{ writeaccess } datatype variablename
{ access-right }{ readaccess }{ writeaccess } datatype variablename
{ readaccess }{ writeaccess } datatype variablename
```

Within a labeled group of declarations, you can override the access on a single line by specifying another access-right keyword with the declaration. The labeled access takes effect again on the following lines.

**Examples**
In these declarations, the instance variables have the access specified by the label that precedes them. Another private variable is defined at the end, where private overrides the public label:

```powerscript
Private:
integer ii_a=10, ii_b=24
string is_Name, is_Address1
Protected:
integer ii_Units
double idb_Results
string is_Lname
Public:
integer ii_Weight
string is_Location="Home"
private integer ii_test
```

Some of these protected declarations have restricted write access:

```powerscript
Protected:
integer ii_Units
privatewrite double idb_Results
privatewrite string is_Lname
```
Declaring constants

Description
Any PowerScript variable declaration of a standard datatype that can be assigned an initial value can be a constant instead of a variable. To make it a constant, include the keyword CONSTANT in the declaration and assign it an initial value.

Syntax
```
CONSTANT { access } datatype constname = value
```

The following table shows the parameters used to declare constants.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT</td>
<td>Declares a constant instead of a variable. The CONSTANT keyword can be before or after the access keywords.</td>
</tr>
<tr>
<td>access (optional)</td>
<td>(For instance variables only) Keywords specifying the access for the constant. For information, see “Access for instance variables” on page 42.</td>
</tr>
<tr>
<td>datatype</td>
<td>A standard datatype for the constant. For decimals, you can include an optional value in brackets to specify the precision of the data. Blobs cannot be constants. For information about PowerBuilder datatypes, see “Standard datatypes” on page 21.</td>
</tr>
<tr>
<td>constname</td>
<td>The name of the constant (must be a valid PowerScript identifier, as described in “Identifier names” on page 5).</td>
</tr>
<tr>
<td>value</td>
<td>A literal or expression of the appropriate datatype that will be the value of the constant. The value is required. For information, see “Initial values for variables” on page 40.</td>
</tr>
</tbody>
</table>

Usage
When declaring a constant, an initial value is required. Otherwise, a compiler error occurs. Assigning a value to a constant after it is declared (that is, redefining a constant in a descendant object) also causes a compiler error.

Examples
Although PowerScript is not case sensitive, these examples of local constants use a convention of capitalizing constant names:

```
constant string LS_HOMECITY = "Boston"
constant real LR_PI = 3.14159265
```
Declaring arrays

Description

An array is an indexed collection of elements of a single datatype. In PowerBuilder, an array can have one or more dimensions. One-dimensional arrays can have a fixed or variable size; multidimensional arrays always have a fixed size. Each dimension of an array can have 2,147,483,647 bytes of elements.

Any simple variable declaration becomes an array when you specify brackets after the variable name. For fixed-size arrays, you specify the sizes of the dimensions inside those brackets.

Syntax

```{ access } datatype variablename { d1, ..., dn } { = { valuelist } }
```

The following table describes the parameters used to declare array variables.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>(For instance variables only) Keywords specifying the access for the variable. For information, see “Access for instance variables” on page 42.</td>
</tr>
<tr>
<td>datatype</td>
<td>The datatype of the variable. You can specify a standard datatype, a system object, or a previously defined structure. For decimals, you can specify the precision of the data by including an optional value in brackets after <code>datatype</code> (see “Syntax of a variable declaration” on page 37): <code>decimal {2} variablename [ ]</code>. For blobs, fixed-length blobs within an array are not supported. If you specify a size after <code>datatype</code>, it is ignored.</td>
</tr>
<tr>
<td>variablename</td>
<td>The name of the variable (name must be a valid PowerScript identifier, as described in “Identifier names” on page 5). You can define additional arrays with the same datatype by naming additional variable names with brackets and optional value lists, separated by commas.</td>
</tr>
</tbody>
</table>
Examples

These declarations create variable-size arrays:

```
integer li_stats[ ]     // Array of integers.
decimal {2} ld_prices[ ] // Array of decimals with
                        // 2 places of precision.
blob lb_data[ ]         // Array of variable-size
                        // blobs.
date ld_birthdays[ ]    // Array of dates.
string ls_city[ ]       // Array of strings.
                        // Each string can be
                        // any length.
```

This statement declares a variable-size array of decimal number (the declaration does not specify a precision, so each element in the array takes the precision of the value assigned to it):

```
dec lc_limit[ ]
```
Declaring arrays

**Fixed arrays**  These declarations create fixed-size, one-dimensional arrays:

  // Each value has 2 digits following the decimal point.
  // Each takes the precision of the value assigned.

**Using TO to change array index values**  These fixed-size arrays use TO to change the range of index values for the array:

- real lr_Rate[2 to 5]  // Array of 4 real numbers:
- integer li_Qty[0 to 2]  // Array of 3 integers
- string ls_Test[-2 to 2]  // Array of 5 strings
- integer li_year[76 to 96]  // Array of 21 integers
- string ls_name[-10 to 15]  // Array of 26 strings

**Incorrect declarations using TO**  In an array dimension, the second number must be greater than the first. These declarations are invalid:

- integer li_count[10 to 5]  // INVALID: 10 is greater than 5
- integer li_price[-10 to -20]  // INVALID: -10 is greater than -20

**Arrays with two or more dimensions**  This declaration creates a six-element, two-dimensional integer array. The individual elements are li_score[1,1], li_score[1,2], li_score[1,3], li_score[2,1], li_score[2,2], and li_score[2,3]:

- integer li_score[2,3]

This declaration specifies that the indexes for the dimensions are 1 to 5 and 10 to 25:

- integer li_RunRate[1 to 5, 10 to 25]

This declaration creates a 3-dimensional 45,000-element array:

- long ll_days[3, 300, 50]
This declaration changes the subscript range for the second and third dimension:

```pascal
integer li_staff[100, 0 to 20, -5 to 5]
```

More declarations of multidimensional arrays:

```pascal
string ls_plant[3,10]  // two-dimensional array
// of 30 strings
dec{2} lc_rate[3,4]   // two-dimensional array of 12
// decimals with 2 digits
// after the decimal point
```

This declaration creates three decimal arrays:

```pascal
decimal{3} lc_first[10], lc_second[15,5], lc_third[
```

### Values for array elements

#### General information

PowerBuilder initializes each element of an array to the same default value as its underlying datatype. For example, in a newly declared integer array:

```pascal
integer li_TaxCode[3]
```


For information about default values for basic datatypes, see “Initial values for variables” on page 40.

#### Simple array

In a simple array, you can override the default values by initializing the elements of the array when you declare the array. You specify the values in a comma-separated list of values enclosed in braces. You do not have to initialize all the elements of the array, but you cannot initialize values in the middle or end without initializing the first elements.

#### Multidimensional array

In a multidimensional array, you still provide the values in a simple, comma-separated list. When the values are assigned to array positions, the first dimension is the fastest-varying dimension, and the last dimension is the slowest-varying. In other words, the values are assigned to array positions by looping over all the values of the first dimension for each value of the second dimension, then looping over all the values of the second dimension for each value of the third, and so on.
Declaring arrays

Assigning values
You can assign values to an array after declaring it using the same syntax of a list of values within braces:

```powershell
integer li_Arr[]
Li_Arr = {1, 2, 3, 4}
```

Examples

Example 1  This statement declares an initialized one-dimensional array of three variables:

```powershell
real lr_Rate[3]={1.20, 2.40, 4.80}
```

Example 2  This statement initializes a two-dimensional array:

```powershell
integer li_units[3,4] = {1,2,3, 1,2,3, 1,2,3, 1,2,3}
```

As a result:

- Li_units[1,1], [1,2], [1,3], and [1,4] are all 1
- Li_units[2,1], [2,2], [2,3], and [2,4] are all 2
- Li_units[3,1], [3,2], [3,3], and [3,4] are all 3

Example 3  This statement initializes the first half of a 3-dimensional array:

```powershell
integer li_units[3,4,2] = &
{1,2,3, 1,2,3, 1,2,3, 1,2,3}
```

As a result:

- Li_units[1,1,1], [1,2,1], [1,3,1], and [1,4,1] are all 1
- Li_units[2,1,1], [2,2,1], [2,3,1], and [2,4,1] are all 2
- Li_units[3,1,1], [3,2,1], [3,3,1], and [3,4,1] are all 3
- Li_units[1,1,2], [1,2,2], [1,3,2], and [1,4,2] are all 0
- Li_units[2,1,2], [2,2,2], [2,3,2], and [2,4,2] are all 0
- Li_units[3,1,2], [3,2,2], [3,3,2], and [3,4,2] are all 0

Size of variable-size arrays

General information
A variable-size array consists of a variable name followed by square brackets but no number. PowerBuilder defines the array elements by use at execution time (subject only to memory constraints). Only one-dimensional arrays can be variable-size arrays.

Because you do not declare the size, you cannot use the TO notation to change the lower bound of the array, so the lower bound of a variable-size array is always 1.
Using arrays with a TO clause in EAServer components
When you generate a proxy for an EAServer component deployed from PowerBuilder that contains an array that uses a TO clause, the proxy object represents the range as a single value because CORBA IDL does not support the TO clause. For example, Int ar1[5 TO 10] is represented as Int ar1[6], with [6] representing the number of array elements. Client applications must declare the array using a single value instead of a range.

How memory is allocated
Initializing elements of a variable-size array allocates memory for those elements. You specify initial values just as you do for fixed-size arrays, by listing the values in braces. The following statement sets code[1] equal to 11, code[2] equal to 242, and code[3] equal to 27. The array has a size of 3 initially, but the size will change if you assign values to higher positions:

```
integer li_code[ ]={11,242,27}
```

For example, these statements declare a variable-size array and assigns values to three array elements:

```
long ll_price[ ]
ll_price[100] = 2000
ll_price[50] = 3000
ll_price[110] = 5000
```

When these statements first execute, they allocate memory as follows:

- The statement `ll_price[100]=2000` will allocate memory for 100 long numbers `ll_price[1]` to `ll_price[100]`, then assign 0 (the default for numbers) to `ll_price[1]` through `ll_price[99]` and assign 2000 to `ll_price[100]`.
- The statement `ll_price[50]=3000` will not allocate more memory but will assign the value 3000 to the 50th element of the `ll_price` array.
- The statement `ll_price[110]=5000` will allocate memory for 10 more long numbers named `ll_price[101]` to `ll_price[110]` and then assign 0 (the default for numbers) to `ll_price[101]` through `ll_price[109]` and assign 5000 to `ll_price[110]`. 

More about arrays

This section provides technical details about:

- Assigning one array to another
- Using array lists to assign values to an array
- Errors that occur when addressing arrays

Assigning one array to another

General information

When you assign one array to another, PowerBuilder uses the following rules to map the values of one onto the other.

One-dimensional arrays

To an unbounded array  The target array is the same as the source:

```powershell
integer a[ ], b[ ]
a = {1,2,3,4}
b = a
```

To a bounded array  If the source array is smaller, values from the source array are copied to the target array and extra values are set to zero. In this example, \( b[5] \) and \( b[6] \) are set to 0:

```powershell
integer a[ ], b[6]
a = {1,2,3,4}
b = a
```

If the source array is larger, values from the source array are copied to the target array until it is full (and extra values from the source array are ignored). In this example, the array \( b \) has only the first three elements of \( a \):

```powershell
integer a[ ], b[3]
a = {1,2,3,4}
b = a
```

Multidimensional arrays

PowerBuilder stores multidimensional arrays in column major order, meaning the first subscript is the fastest varying—\([1,1],[2,1],[3,1]\).

When you assign one array to another, PowerBuilder linearizes the source array in column major order, making it a one-dimensional array. PowerBuilder then uses the rules for one-dimensional arrays (described above) to assign the array to the target.

Not all array assignments are allowed, as described in the following rules.
**One multidimensional array to another**  If the dimensions of the two arrays match, the target array becomes an exact copy of the source:

```plaintext
integer a[2,10], b[2,10]
    a = b
```

If both source and target are multidimensional but do not have matching dimensions, the assignment is not allowed and the compiler reports an error:

```plaintext
integer a[2,10], b[4,10]
    a = b // Compiler error
```

**One-dimensional array to a multidimensional array**  A one-dimensional array can be assigned to a multidimensional array. The values are mapped onto the multidimensional array in column major order:

```plaintext
integer a[], b[2,2]
    b = a
```

**Multidimensional array to a one-dimensional array**  A multidimensional array can also be assigned to a one-dimensional array. The source is linearized in column major order and assigned to the target:

```plaintext
integer a[], b[2,2]
    a = b
```

### Examples

Suppose you declare three arrays (a, b, and c). One (c) is unbounded and one-dimensional; the other two (a and b) are multidimensional with different dimensions:

```plaintext
integer c[], a[2,2], b[3,3] = {1,2,3,4,5,6,7,8,9}
```

Array b is laid out like this:

<table>
<thead>
<tr>
<th></th>
<th>b[1,1]</th>
<th>b[1,2]</th>
<th>b[1,3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

This statement causes a compiler error, because a and b have different dimensions:

```plaintext
    a = b // Compiler error
```

This statement explicitly linearizes b into c:

```plaintext
    c = b
```

You can then assign the linearized version of the array to a:

```plaintext
    a = c
```
Declaring arrays

The values in array \( a \) are laid out like this:

\[
\begin{array}{c|c}
1 & 3 \\
2 & 4 \\
\end{array}
\]

Initializing \( a \) with an arraylist produces the same result:

\[
\text{integer } a[2,2] = \{1,2,3,4\}
\]

The following section describes arraylists.

Using arraylists to assign values to an array

General information
In PowerBuilder, an arraylist is a list of values enclosed in braces used to initialize arrays. An arraylist represents a one-dimensional array, and its values are assigned to the target array using the rules for assigning arrays described in “Assigning one array to another” on page 54.

Examples
In this declaration, a variable-size array is initialized with four values:

\[
\text{integer } a[ ] = \{1,2,3,4\}
\]

In this declaration, a fixed-size array is initialized with four values (the rest of its values are zeros):

\[
\text{integer } a[10] = \{1,2,3,4\}
\]

In this declaration, a fixed-size array is initialized with four values. Because the array’s size is set at 4, the rest of the values in the arraylist are ignored:

\[
\text{integer } a[4] = \{1,2,3,4,5,6,7,8\}
\]

In this declaration, values 1, 2, and 3 are assigned to the first column and the rest to the second column:

\[
\text{integer } a[3,2] = \{1,2,3,4,5,6\}
\]

\[
\begin{array}{c|c}
1 & 4 \\
2 & 5 \\
3 & 6 \\
\end{array}
\]

If you think of a three-dimensional array as having pages of rows and columns, then the first column of the first page has the values 1 and 2, the second column on the first page has 3 and 4, and the first column on the second page has 5 and 6.
Errors that occur when addressing arrays

**Fixed-size arrays**

In PowerBuilder, referring to array elements outside the declared size causes an error at runtime; for example:

```plaintext
int test[10]
test[0]=50   // This causes an execution error.
int trial[5,10]
trial[6,2]=75 // This causes an execution error.
trial[4,11]=75 // This causes an execution error.
```

**Variable-size arrays**

Assigning a value to an element of a variable-size array that is outside its current values increases the array’s size. However, accessing a variable-size array above its largest assigned value or below its lower bound causes an error at runtime:

```plaintext
integer li_stock[ ]
li_stock[50]=200
    // Establish array size 50 elements.
IF li_stock[51]=0 then Beep(1)
    // This causes an execution error.
IF li_stock[0]=0 then Beep(1)
    // This causes an execution error.
```

Declaring external functions

**Description**

External functions are functions written in languages other than PowerScript and stored in dynamic link libraries. On Windows, dynamic libraries have the extension `DLL`. If you deploy a component written in PowerBuilder to a UNIX server, the dynamic libraries it calls have the extension `.so`, `.sl`, or `.a`, depending on the UNIX operating system. You can use external functions that are written in any language that supports dynamic libraries.
Before you can use an external function in a script, you must declare it as one of two types:

- **Global external functions** These are available anywhere in the application.
- **Local external functions** These are defined for a particular type of window, menu, user object, or user-defined function. These functions are part of the object’s definition and can always be used in scripts for the object itself. You can also choose to make these functions accessible to other scripts.

To understand how to declare and call an external function, see the documentation from the developer of the external function library.

### Syntax

**External function syntax** Use the following syntax to declare an external function:

```plaintext
{ access } FUNCTION returndatatype name ( { { REF } datatype1 arg1, ...
    ... ( REF ) datatypen argn ) LIBRARY "libname"
    ALIAS FOR "extname[ansi]"
```

**External subroutine syntax** To declare external subroutines (which are the same as external functions except that they do not return a value), use this syntax:

```plaintext
{ access } SUBROUTINE name ( { { REF } datatype1 arg1, ...
    ... ( REF ) datatypen argn ) LIBRARY "libname"
    ALIAS FOR "extname[ansi]"
```

The following table describes the parameters used to declare external functions and subroutines:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>access (optional)</td>
<td>(Local external functions only) Public, Protected, or Private specifies the access level of a local external function. The default is Public. For more information, see the section about specifying access of local functions in &quot;Usage&quot; next.</td>
</tr>
<tr>
<td>FUNCTION or SUBROUTINE</td>
<td>A keyword specifying the type of call, which determines the way return values are handled. If there is a return value, declare it as a FUNCTION; if it returns nothing or returns VOID, specify SUBROUTINE.</td>
</tr>
<tr>
<td>returndatatype</td>
<td>The datatype of the value returned by the function.</td>
</tr>
</tbody>
</table>
Usage

Specifying access of local functions When declaring a local external function, you can specify its access level—which scripts have access to the function.
Declaring external functions

The following table describes where local external functions can be used when they are declared with a given access level:

<table>
<thead>
<tr>
<th>Access level</th>
<th>Where you can use the local external function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Any script in the application.</td>
</tr>
<tr>
<td>Private</td>
<td>Scripts for events in the object for which the function is declared. You cannot use the function in descendants of the object.</td>
</tr>
<tr>
<td>Protected</td>
<td>Scripts for the object for which the function is declared and its descendants.</td>
</tr>
</tbody>
</table>

Use of the access keyword with local external functions works the same as the access-right keywords for instance variables.

To be available to a PowerBuilder application running on any Windows platform, the DLL must be in one of the following directories:

- The current directory
- The Windows directory
- The Windows System subdirectory
- Directories on the DOS path

If you are deploying a PowerBuilder custom class user object as an EAServer component, you must make sure any dynamic library it references is available on the server. If you do not specify the location of the library when you declare it, make sure it is installed in an accessible location:

- On a Windows server, the DLL must be in the application path of the server’s executable file.
- On a UNIX server, the location of the shared library must be listed in the server’s library path environment variable (for example, LD_LIBRARY_PATH on Solaris) or the library must be in the lib directory of the EAServer installation.

Examples

In the examples application that comes with PowerBuilder, external functions are declared as local external functions in a user object called u_external_function_win32. The scripts that call the functions are user object functions, but because they are part of the same user object, you do not need to use object notation to call them.
Example 1  These declarations allow PowerBuilder to call the functions required for playing a sound in the WINMM.DLL:

```powerbuilder
//playsound
FUNCTION boolean sndPlaySoundA (string SoundName, 
   uint Flags) LIBRARY "WINMM.DLL" ALIAS FOR 
   "sndPlaySoundA;ansi"
FUNCTION uint waveOutGetNumDevs () LIBRARY "WINMM.DLL"
```

A function called uf_playsound in the examples application provided with PowerBuilder calls the external functions. Uf_playsound is called with two arguments (as_filename and ai_option) that are passed through to sndPlaySoundA.

Values for ai_option are as defined in the Windows documentation, as commented here:

```powerbuilder
//Options as defined in mmystem.h.
//These may be or'd together.
//#define SND_SYNC 0x0000
//play synchronously (default)
//#define SND_ASYNC 0x0001
//play asynchronously
//#define SND_NODEFAULT 0x0002
//do not use default sound
//#define SND_MEMORY 0x0004
//lpszSoundName points to a memory file
//#define SND_LOOP 0x0008
//loop the sound until next sndPlaySound
//#define SND_NOSTOP 0x0010
//do not stop any currently playing sound
```

```powerbuilder
uint lui_numdevs

lui_numdevs = WaveOutGetNumDevs()
IF lui_numdevs > 0 THEN
   sndPlaySoundA(as_filename,ai_option)
   RETURN 1
ELSE
   RETURN -1
END IF
```

Example 2  This is the declaration for the Windows GetSysColor function:

```powerbuilder
FUNCTION ulong GetSysColor (int index) LIBRARY "USER32.DLL"
```
Declaring external functions

This statement calls the external function. The meanings of the index argument and the return value are specified in the Windows documentation:

```c
RETURN GetSysColor (ai_index)
```

**Example 3** This is the declaration for the Windows `GetSysColor` function:

```c
FUNCTION int GetSystemMetrics (int index) LIBRARY "USER32.DLL"
```

These statements call the external function to get the screen height and width:

```c
RETURN GetSystemMetrics(1)
RETURN GetSystemMetrics(0)
```

Datatypes for external function arguments

When you declare an external function in PowerBuilder, the datatypes of the arguments must correspond with the datatypes as declared in the function’s source definition. This section documents the correspondence between datatypes in external functions and datatypes in PowerBuilder. It also includes information on byte alignment when passing structures by value.

Use the tables to find out what PowerBuilder datatype to use in an external function declaration. The PowerBuilder datatype you select depends on the datatype in the source code for the function. The first column lists datatypes in source code. The second column describes the datatype so you know exactly what it is. The third column lists the PowerBuilder datatype you should use in the external function declaration.

<table>
<thead>
<tr>
<th>Datatype in source code</th>
<th>Size, sign, precision</th>
<th>PowerBuilder datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(any pointer)</em></td>
<td>32-bit pointer</td>
<td>Long</td>
</tr>
<tr>
<td>char *</td>
<td>Array of bytes of variable length</td>
<td>Blob</td>
</tr>
</tbody>
</table>

Windows 32-bit FAR pointers, such as LPBYTE, LPDWORD, LPINT, LPLONG, LPVOID, and LPWORD, are declared in PowerBuilder as long datatypes. HANDLE is defined as 32 bits unsigned and is declared in PowerBuilder as an UnsignedLong.

Near-pointer datatypes (such as PSTR and NPSTR) are not supported in PowerBuilder.
Table 3-10: PowerBuilder datatypes for characters and strings

<table>
<thead>
<tr>
<th>Datatype in source code</th>
<th>Size, sign, precision</th>
<th>PowerBuilder datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8 bits, signed</td>
<td>Char</td>
</tr>
<tr>
<td>string</td>
<td>32-bit pointer to a null-terminated array of bytes of variable length</td>
<td>String</td>
</tr>
</tbody>
</table>

The Windows 32-bit FAR pointer LPSTR is declared in PowerBuilder as string.

Reference arguments

When you pass a string to an external function by reference, all memory management is done in PowerBuilder. The string variable must be long enough to hold the returned value. To ensure that this is true, first declare the string variable, and then use the Space function to fill the variable with blanks equal to the maximum number of characters that you expect the function to return.

Table 3-11: PowerBuilder datatypes for fixed-point values

<table>
<thead>
<tr>
<th>Datatype in source code</th>
<th>Size, sign, precision</th>
<th>PowerBuilder datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8 bits, unsigned</td>
<td>Byte</td>
</tr>
<tr>
<td>short</td>
<td>16 bits, signed</td>
<td>Integer</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16 bits, unsigned</td>
<td>UnsignedInteger</td>
</tr>
<tr>
<td>int</td>
<td>32 bits, signed</td>
<td>Long</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32 bits, unsigned</td>
<td>UnsignedLong</td>
</tr>
<tr>
<td>long</td>
<td>32 bits, signed</td>
<td>Long</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32 bits, unsigned</td>
<td>UnsignedLong</td>
</tr>
<tr>
<td>longlong</td>
<td>64 bits, signed</td>
<td>LongLong</td>
</tr>
</tbody>
</table>

The Windows definition WORD is declared in PowerBuilder as UnsignedInteger and the Windows definition DWORD is declared as an UnsignedLong. You cannot call external functions with return values or arguments of type short.

Table 3-12: PowerBuilder datatypes for floating-point values

<table>
<thead>
<tr>
<th>Datatype in source code</th>
<th>Size, sign, precision</th>
<th>PowerBuilder datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32 bits, single precision</td>
<td>Real</td>
</tr>
<tr>
<td>double</td>
<td>64 bits, double precision</td>
<td>Double</td>
</tr>
</tbody>
</table>

PowerBuilder does not support 80-bit doubles on Windows.
Declaring external functions

Date and time
The PowerBuilder datatypes Date, DateTime, and Time are structures and have no direct equivalent for external functions in C.

Passing structures by value
You can pass PowerBuilder structures to external C functions if they have the same definitions and alignment as the structure’s components. The DLL or shared library must be compiled using byte alignment; no padding is added to align fields within the structure.

Calling external functions

Global external functions
In PowerBuilder, you call global external functions using the same syntax as for calling user-defined global and system functions. As with other global functions, global external functions can be triggered or posted but not called dynamically.

Local external functions
Call local functions using the same syntax as for calling object functions. They can be triggered or posted and called dynamically.

For information
For information, see “Syntax for calling PowerBuilder functions and events” on page 111.

Defining source for external functions
You can use external functions written in any language that supports the standard calling sequence for 32-bit platforms. If you are calling functions on Windows in libraries that you have written yourself, remember that you need to export the functions. Depending on your compiler, you can do this in the function prototype or in a linker definition (.DEF) file. For more information about using external functions, see Application Techniques.

Use _stdcall convention
C and C++ compilers typically support several calling conventions, including _cdecl (the default calling convention for C programs), _stdcall (the standard convention for Windows API calls), _fastcall, and _thiscall. PowerBuilder, like many other Windows development tools, requires external functions to be exported using the WINAPI (_stdcall) format. Attempting to use a different calling convention can cause an application crash.

When you create your own C or C++ DLLs containing functions to be used in PowerBuilder, make sure that they use the standard convention for Windows API calls.
For example, if you are using a DEF file to export function definitions, you can declare the function like this:

```c
LONG WINAPI myFunc()
{
    ...
}
```

### Declaring DBMS stored procedures as remote procedure calls

**Description**
In PowerBuilder, you can use dot notation for calling non-result-set stored procedures as remote procedure calls (RPCs):

```c
object.function
```

You can call database procedures in Sybase, Oracle, Informix, and other ODBC databases with stored procedures.

RPCs provide support for Oracle PL/SQL tables and parameters that are defined as both input and output. You can call overloaded procedures.

**Syntax**

<table>
<thead>
<tr>
<th>Transaction object</th>
</tr>
</thead>
</table>
| FUNCTION rtndatatype functionname ( { REF } datatype1 arg1,....,
  { REF } datatype2 argn ) RPCFUNC ( ALIAS FOR "spname" ) |
| SUBROUTINE functionname ( { REF } datatype1 arg1,....,
  { REF } datatype2 argn ) RPCFUNC ( ALIAS FOR "spname" ) |

**Table 3-13: RPC declaration parameters**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION or SUBROUTINE</td>
<td>A keyword specifying the type of call, which determines the way return values are handled. If there is a return value, declare it as a FUNCTION. If it returns nothing or returns VOID, specify SUBROUTINE.</td>
</tr>
<tr>
<td>rtndatatype</td>
<td>In a FUNCTION declaration, the datatype of the value returned by the function.</td>
</tr>
<tr>
<td>functionname</td>
<td>The name of the database procedure as you will call it in PowerBuilder. If the name in the DBMS is different, use ALIAS FOR to associate the DBMS name with the PowerBuilder name.</td>
</tr>
</tbody>
</table>
Declaring DBMS stored procedures as remote procedure calls

**Usage**

If a function does not return a value (for example, it returns Void), specify the declaration as a subroutine instead of a function.

RPC declarations are always associated with a transaction object. You declare them as local external functions. The Declare Local External Functions dialog box has a Procedures button (if the connected database supports stored procedures), which gives you access to a list of stored procedures in the database.

For more information, see *Application Techniques*.

**Examples**

**Example 1** This declaration of the GIVE_RAISE_PROC stored procedure is declared in the User Object painter for a transaction object (the declaration appears on one line):

```powerbuilder
FUNCTION double GIVE_RAISE(ref double SALARY) RPCFUNC ALIAS FOR "GIVE_RAISE_PROC"
```
This code calls the function in a script:

```plaintext
double val = 20000
double rv
rv = SQLCA.give_raise(val)
```

**Example 2** This declaration for the stored procedure SPM8 does not need an ALIAS FOR phrase, because the PowerBuilder and DBMS names are the same:

```plaintext
FUNCTION integer SPM8(integer value) RPCFUNC
```

This code calls the SPM8 stored procedure:

```plaintext
int myresult
myresult = SQLCA.spm8(myresult)
IF SQLCA.sqlcode <> 0 THEN
    messagebox("Error", SQLCA.sqlerrtext)
END IF
```
Declaring DBMS stored procedures as remote procedure calls
CHAPTER 4

Operators and Expressions

About this chapter
This chapter describes the operators supported in PowerScript and how to use them in expressions.

Contents
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators in PowerBuilder</td>
<td>69</td>
</tr>
<tr>
<td>Operator precedence in PowerBuilder expressions</td>
<td>74</td>
</tr>
<tr>
<td>Datatype of PowerBuilder expressions</td>
<td>75</td>
</tr>
</tbody>
</table>

Operators in PowerBuilder

General information
Operators perform arithmetic calculations; compare numbers, text, and boolean values; execute relational operations on boolean values; and concatenate strings and blobs.

Three types
PowerScript supports three types of operators:
• Arithmetic operators for numeric datatypes
• Relational operators for all datatypes
• Concatenation operator for string datatypes

Operators used in DataWindow objects
The documentation for DataWindows describes how operators are used in DataWindow expressions.
Arithmetic operators in PowerBuilder

The following table lists the arithmetic operators used in PowerBuilder.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Total=SubTotal+Tax</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Price=Price-Discount</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unless you have prohibited the use of dashes in identifier names, you must surround the minus sign with spaces.</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Total=Quantity*Price</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Factor=Discount/Price</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation</td>
<td>Rank=Rating^2.5</td>
</tr>
</tbody>
</table>

**Usage**

**Operator shortcuts for assignments**  For information about shortcuts that combine arithmetic operators with assignments (such as ++ and +=), see Assignment on page 120.

**Subtraction**  If the option Allow Dashes in Identifiers is checked on the Script tab in the Options dialog box, you must always surround the subtraction operator and the -- operator with spaces. Otherwise, PowerBuilder interprets the expression as an identifier.

For information about dashes in identifiers, see “Identifier names” on page 5.

**Multiplication and division**  Multiplication and division are carried out to full precision (16–28 digits). Decimal numbers are rounded (not truncated) on assignment.

**Calculation with NULL**  When you form an arithmetic expression that contains a NULL value, the expression’s value is null. Thinking of null as *undefined* makes this easier to understand.

For more information about null values, see “NULL values” on page 8.

**Errors and overflows**  The following problems can occur when using arithmetic operators:

- Division by zero, exponentiation of negative values, and so on cause errors at runtime.
- Overflow of real, double, and decimal values causes errors at runtime.
Overflow of signed or unsigned integers and longs causes results to wrap. However, because integers are promoted to longs in calculations, wrapping does not occur until the result is explicitly assigned to an integer variable.

For more information about type promotion, see “Datatype of PowerBuilder expressions” on page 75.

Examples

Subtraction  This statement always means subtract B from A:

\[ A - B \]

If DashesInIdentifiers is set to 1, the following statement means a variable named A-B, but if DashesInIdentifiers is set to 0, it means subtract B from A:

\[ A - B \]

Precision for division  These examples show the values that result from various operations on decimal values:

```plaintext
decimal {4} a,b,d,e,f
decimal {3} c
a = 20.0/3       // a contains  6.6667
b = 3 * a       // b contains 20.0001
c = 3 * a       // c contains 20.000
```

```plaintext
d = 3 * (20.0/3)       // d contains 20.0000
e = Truncate(20.0/3, 4) // e contains  6.6666
f = Truncate(20.0/3, 5) // f contains  6.6667
```

Calculations with null  When the value of variable \( c \) is null, the following assignment statements all set the variable \( a \) to null:

```plaintext
integer a, b=100, c

SetNULL(c)

a = b+c       // all statements set a to NULL
a = b - c
a = b*c
a = b/c
```

Overflow  This example illustrates the value of the variable \( i \) after overflow occurs:

```plaintext
integer i
i = 32767
i = i + 1       // i is now -32768
```
Relational operators in PowerBuilder

Description

PowerBuilder uses relational operators in boolean expressions to evaluate two or more operands. Logical operators can join relational expressions to form more complex boolean expressions.

The result of evaluating a boolean expression is always true or false.

The following table lists relational and logical operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equals</td>
<td>if Price=100 then Rate=.05</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>if Price&gt;100 then Rate=.05</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>if Price&lt;100 then Rate=.05</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
<td>if Price&lt;&gt;100 then Rate=.05</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td>if Price&gt;=100 then Rate=.05</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>if Price&lt;=100 then Rate=.05</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical negation</td>
<td>if NOT Price=100 then Rate=.05</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
<td>if Tax&gt;3 AND Ship &lt;5 then Rate=.05</td>
</tr>
<tr>
<td>OR</td>
<td>Logical or</td>
<td>if Tax&gt;3 OR Ship&lt;5 then Rate=.05</td>
</tr>
</tbody>
</table>

Usage

Comparing strings

When PowerBuilder compares strings, the comparison is case sensitive. Trailing blanks are significant.

For information on comparing strings regardless of case, see the functions Upper on page 1152 and Lower on page 725.

To remove trailing blanks, use the RightTrim function. To remove leading blanks, use the LeftTrim function. To remove leading and trailing blanks, use the Trim function. For information about these functions, see RightTrim on page 919, LeftTrim on page 695, and Trim on page 1140.

Decimal operands

Relational operators that operate on numeric values (including =, >, <, <=, and >=) can take decimal operands. The precision of the decimal operand is maintained in comparisons.

Null value evaluations

When you form a boolean expression that contains a null value, the AND and OR operators behave differently. Thinking of null as undefined (neither true nor false) makes the results easier to calculate.

For more information about null values, see “NULL values” on page 8.
Examples

Case-sensitive comparisons  If you compare two strings with the same text but different case, the comparison fails. But if you use the Upper or Lower function, you can ensure that the case of both strings are the same so that only the content affects the comparison:

```plaintext
City1 = "Austin"
City2 = "AUSTIN"
IF City1 = City2 ...  // Returns FALSE

City1 = "Austin"
City2 = "AUSTIN"
IF Upper(City1) = Upper(City2)... // Returns TRUE
```

Trailing blanks in comparisons  In this example, trailing blanks in one string cause the comparison to fail:

```plaintext
City1 = "Austin"
City2 = "Austin       
IF City1 = City2 ...  // Returns FALSE
```

Logical expressions with null values  In this example, the expressions involving the variable `f`, which has been set to null, have null values:

```plaintext
boolean d, e = TRUE, f
SetNull(f)
d = e and f   // d is NULL
d = e or f    // d is TRUE
```

Concatenation operator in PowerBuilder

Description  The PowerBuilder concatenation operator joins the contents of two variables of the same type to form a longer value. You can concatenate strings and blobs. The following table shows the concatenation operator.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Concatenate</td>
<td>&quot;cat &quot; + &quot;dog&quot;</td>
</tr>
</tbody>
</table>

Examples

Example 1  These examples concatenate several strings:

```plaintext
string Test
Test = "over" + "stock" // Test contains "overstock"
string Lname, Fname, FullName
FullName = Lname + ', ', Fname
// FullName contains last name and first name,
// separated by a comma and space.
```
Example 2  This example shows how a blob can act as an accumulator when reading data from a file:

```
integer i, fnum, loops
blob tot_b, b
.
FOR i = 1 to loops
  bytes_read = FileRead(fnum, b)
  tot_b = tot_b + b
NEXT
```

Operator precedence in PowerBuilder expressions

**Order of precedence**

To ensure predictable results, all operators in a PowerBuilder expression are evaluated in a specific order of precedence. When the operators have the same precedence, PowerBuilder evaluates them left to right.

These are the operators in descending order of precedence:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Grouping (see note below on overriding)</td>
</tr>
<tr>
<td>+, -</td>
<td>Unary plus and unary minus (indicates positive or negative number)</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>*, /</td>
<td>Multiplication and division</td>
</tr>
<tr>
<td>+, -</td>
<td>Addition and subtraction; string concatenation</td>
</tr>
<tr>
<td>=, &gt;, &lt;, &lt;=, &gt;=, &lt;&gt;</td>
<td>Relational operators</td>
</tr>
<tr>
<td>NOT</td>
<td>Negation</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
</tr>
<tr>
<td>OR</td>
<td>Logical or</td>
</tr>
</tbody>
</table>

**How to override**

To override the order, enclose expressions in parentheses. This identifies the group and order in which PowerBuilder will evaluate the expressions. When there are nested groups, the groups are evaluated from the inside out.

For example, in the expression \((x + (y \ast (a + b)))\), \(a + b\) is evaluated first. The sum of \(a\) and \(b\) is then multiplied by \(y\), and this product is added to \(x\).
Datatype of PowerBuilder expressions

General information
The datatype of an expression is important when it is the argument for a function or event. The expression’s datatype must be compatible with the argument’s definition. If a function is overloaded, the datatype of the argument determines which version of the function to call.

There are three types: numeric, string, and char datatypes.

Numeric datatypes in PowerBuilder

General information
All numeric datatypes are compatible with each other.

What PowerBuilder does
PowerBuilder converts datatypes as needed to perform calculations and make assignments. When PowerBuilder evaluates a numeric expression, it converts the datatypes of operands to datatypes of higher precedence according to the operators and the datatypes of other values in the expression.

Datatype promotion when evaluating numeric expressions

Order of precedence
The PowerBuilder numeric datatypes are listed here in order of highest to lowest precedence (the order is based on the range of values for each datatype):

- Double
- Real
- Decimal
- LongLong
- UnsignedLong
- Long
- UnsignedInteger
- Integer
- Byte

Rules for type promotion

Datatypes of operands
If operands in an expression have different datatypes, the value whose type has lower precedence is converted to the datatype with higher precedence.

Unsigned versus signed
Unsigned has precedence over signed, so if one operand is signed and the other is unsigned, both are promoted to the unsigned version of the higher type. For example, if one operator is a long and another UnsignedInteger, both are promoted to UnsignedLong.
Datatype of PowerBuilder expressions

Operators The effects of operators on an expression’s datatype are:

- +, -, *  The minimum precision for addition, subtraction, and multiplication calculations is `long`. Integer types are promoted to `long` types before doing the calculation and the expression’s resulting datatype is, at a minimum, `long`. When operands have datatypes of higher precedence, other operands are promoted to match based on the Datatypes of operands rule above.

- / and ^  The minimum precision for division and exponentiation is `double`. All types are promoted to `double` before doing the calculation, and the expression’s resulting datatype is `double`.

- Relational  Relational operators do not cause promotion of numeric types.

Datatypes of literals When a literal is an operand in an expression, its datatype is determined by the literal’s value. The datatype of a literal affects the type promotion of the literal and other operands in an expression.

Table 4-5: Datatypes of literal operands in an expression

<table>
<thead>
<tr>
<th>Literal</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer literals (no decimal point or exponent) within the range of Long</td>
<td>Long</td>
</tr>
<tr>
<td>Integer literals beyond the range of Long and within the range of UnsignedLong</td>
<td>UnsignedLong</td>
</tr>
<tr>
<td>Integer literals beyond the range of UnsignedLong and within the range of LongLong</td>
<td>UnsignedLong</td>
</tr>
<tr>
<td>Numeric literals with a decimal point (but no exponent)</td>
<td>Decimal</td>
</tr>
<tr>
<td>Numeric literals with a decimal point and explicit exponent</td>
<td>Double</td>
</tr>
</tbody>
</table>

Out of range

Integer literals beyond the range of LongLong cause compiler errors.
Assignment and datatypes

Assignment is not part of expression evaluation. In an assignment statement, the value of an expression is converted to the datatype of the left-hand variable. In the expression

\[ c = a + b \]

the datatype of \( a+b \) is determined by the datatypes of \( a \) and \( b \). Then, the result is converted to the datatype of \( c \).

Even when PowerBuilder performs a calculation at high enough precision to handle the results, assignment to a lower precision variable can cause overflow, producing the wrong result.

**Example 1**  Consider this code:

```powerbuilder
integer a = 32000, b = 1000
long d

d = a + b
```

The final value of \( d \) is 33000. The calculation proceeds like this:

Convert integer \( a \) to long
Convert integer \( b \) to long
Add the longs \( a \) and \( b \)
Assign the result to the long \( d \)

Because the variable \( d \) is a long, the value 33000 does not cause overflow.

**Example 2**  In contrast, consider this code with an assignment to an integer variable:

```powerbuilder
integer a = 32000, b = 1000, c
long e

c = a + b

e = c
```

The resulting value of \( c \) and \( e \) is -32536. The calculation proceeds like this:

Add the integers \( a \) and \( b \)
Assign the result to \( c \)
Convert integer \( c \) to long and assign the result to \( e \)

The assignment to the integer variable \( c \) causes the long result of the addition to be truncated, causing overflow and wrapping. Assigning \( c \) to \( e \) cannot restore the lost information.
Datatype of PowerBuilder expressions

String and char datatypes in PowerBuilder

General information

There is no explicit char literal type.

String literals convert to type char using the following rules:

- When a string literal is assigned to a char variable, the first character of the string literal is assigned to the variable. For example:

  ```
  char c = "xyz"
  ```

  results in the character x being assigned to the char variable c.

- Special characters (such as newline, formfeed, octal, hex, and so on) can be assigned to char variables using string conversion, such as:

  ```
  char c = "-n"
  ```

String variables assigned to char variables also convert using these rules. A char variable assigned to a string variable results in a one-character string.

Assigning strings to char arrays

As with other datatypes, you can use arrays of chars. Assigning strings to char arrays follows these rules:

- If the char array is unbounded (defined as a variable-size array), the contents of the string are copied directly into the char array.

- If the char array is bounded and its length is less than or equal to the length of the string, the string is truncated in the array.

- If the char array is bounded and its length is greater than the length of the string, the entire string is copied into the array along with its zero terminator. Remaining characters in the array are undetermined.

Assigning char arrays to strings

When a char array is assigned to a string variable, the contents of the array are copied into the string up to a zero terminator, if found, in the char array.

Using both strings and chars in an expression

Expressions using both strings and char arrays promote the chars to strings before evaluation. For example, the following promotes the contents of c to a string before comparison with the string “x”:

```
char c
.
.
if (c = "x") then
```

Using chars in PowerScript functions

All PowerScript functions that take strings also take chars and char arrays, subject to the conversion rules described above.
This chapter describes basic concepts for structures and objects and how you define, declare, and use them in PowerScript.

### Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About structures</td>
<td>79</td>
</tr>
<tr>
<td>About objects</td>
<td>80</td>
</tr>
<tr>
<td>Assignment for objects and structures</td>
<td>86</td>
</tr>
</tbody>
</table>

### About structures

#### General information

A structure is a collection of one or more variables (sometimes called elements) that you want to group together under a single name. The variables can have any datatype, including standard and object datatypes and other structures.

#### Defining structures

When you define a structure in the Structure painter or an object painter (such as Window, Menu, or User Object), you are creating a structure definition. To use the structure, you must declare it. When you declare it, an instance of it is automatically created for you. When it goes out of scope, the structure is destroyed.

For details about defining structures, see the PowerBuilder *Users Guide*.

#### Declaring structures

If you have defined a global structure in the Structure painter called `str_emp_data`, you can declare an instance of the structure in a script or in an object’s instance variables. If you define the structure in an object painter, you can only declare instances of the structure in the object’s instance variables and scripts.

This declaration declares two instances of the structure `str_emp_data`:

```
str_emp_data str_emp1, str_emp2
```
About objects

In object-oriented programming, an object is a self-contained module containing state information and associated methods. Most entities in PowerBuilder are objects: visual objects such as windows and controls on windows, nonvisual objects such as transaction and error objects, and user objects that you design yourself.

An object class is a definition of an object. You create an object’s definition in the appropriate painter: Window, Menu, Application, Structure, or User Object painter. In the painter, you add controls to be part of the object, specify initial values for the object’s properties, define its instance variables and functions, and write scripts for its events and functions.

Referred to structure variables

In scripts, you refer to the structure’s variables using dot notation:

`structurename.variable`

These statements assign values to the variables in `str_emp_data`:

```plaintext
str_emp1.emp_id = 100
str_emp1.emp_lname = "Jones"
str_emp1.emp_salary = 200

str_emp2.emp_id = 101
str_emp2.emp_salary = str_emp1.salary * 1.05
```

Using structures as instance variables

If the structure is declared as part of an object, you can qualify the structure name using dot notation:

`objectname.structurename.variable`

Suppose that this declaration is an instance variable of the window `w_customer`:

```plaintext
str_cust_data str_cust1
```

The following statement in a script for the object refers to a variable of `str_cust_data`. The pronoun `This` is optional, because the structure declaration is part of the object:

```plaintext
This.str_cust1.name
```

The following statement in a script for some other object qualifies the structure with the window name:

```plaintext
w_customer.str_cust1.name
```
An object instance is an occurrence of the object created during the execution of your application. Your code instantiates an object when it allocates memory for the object and defines the object based on the definition in the object class.

An object reference is your handle to the object instance. To interact with an object, you need its object reference. You can assign an object reference to a variable of the appropriate type.

There are two categories of objects supported by PowerBuilder: system objects (also referred to as system classes) defined by PowerBuilder and user objects you define in painters.

**System objects** The PowerBuilder system objects or classes are inherited from the base class PowerObject. The system classes are the ancestors of all the objects you define. To see the system class hierarchy, select the System tab in the Browser, select PowerObject, and select Show Hierarchy and Expand All from the pop-up menu.

**User objects** You can create user object class definitions in several painters: Window, Menu, Application, Structure, and User Object painters. The objects you define are inherited from one of the system classes or another of your classes.

Some painters use many classes. In the Window and User Object painters, the main definition is inherited from the window or user object class. The controls you use are also inherited from the system class for that control.

**About user objects**

**Two types**

There are two major types of user objects: visual and class.

**Visual user objects**

A visual user object is a reusable control or set of controls that has a certain behavior. There are three types—standard, custom, and external.
### Table 5-1: Visual user object types

<table>
<thead>
<tr>
<th>Visual user objects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>Inherited from a specific visual control. You can set properties and write scripts so that the control is ready for use. It has the same events and properties as the control it is inherited from plus any that you add.</td>
</tr>
<tr>
<td>Custom</td>
<td>Inherited from the UserObject system class. You can include many controls in the user object and write scripts for their events. Each control in the user object has the same events and properties as the controls from which they are inherited plus any that you add.</td>
</tr>
<tr>
<td>External</td>
<td>A user object that displays a visual control defined in a DLL. The control is not part of the PowerBuilder object hierarchy. The DLL developer provides information for setting style bits that control its presentation. Its events, functions, and properties are specified by the developer of the DLL. An external user object is not the same as an OCX, which you can put in an OLE control.</td>
</tr>
</tbody>
</table>

### Class user objects

Class user objects consist of properties, functions, and sometimes events. They have no visual component. There are two types—standard and custom.

### Table 5-2: Class user object types

<table>
<thead>
<tr>
<th>Class user objects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>Inherits its definition from a nonvisual PowerBuilder object, such as the Transaction or Error object. You can add instance variables and functions. A few nonvisual objects have events—to write scripts for these events, you have to define a class user object.</td>
</tr>
<tr>
<td>Custom</td>
<td>An object of your own design for which you define instance variables, events, and functions in order to encapsulate application-specific programming in an object.</td>
</tr>
</tbody>
</table>

For information on defining and using user objects, see the PowerBuilder Users Guide.
Instantiating objects

Because of the way PowerBuilder object classes and instances are named, it is easy to think they are the same thing. For example, when you define a window in the Window painter, you are defining an object class.

One instance

When you open a window with the simplest format of the Open function, you are instantiating an object instance. Both the class definition and the instance have the same name. In your application, \texttt{w\_main} is a global variable of type \texttt{w\_main}:

\begin{verbatim}
    Open(w\_main)
\end{verbatim}

When you open a window this way, you can only open one instance of the object.

Several instances

If you want to open more than one instance of a window class, you need to define a variable to hold each object reference:

\begin{verbatim}
    w\_main \ w\_1, \ w\_2
    Open(w\_1)
    Open(w\_2)
\end{verbatim}

You can also open windows by specifying the class in the \texttt{Open} function:

\begin{verbatim}
    window \ w\_1, \ w\_2
    Open(w\_1, "w\_main")
    Open(w\_2, "w\_main")
\end{verbatim}

For class user objects, you always define a variable to hold the object reference and then instantiate the object with the \texttt{CREATE} statement:

\begin{verbatim}
    uo\_emp\_data \ uo\_1, \ uo\_2
    uo\_1 = CREATE uo\_emp\_data
    uo\_2 = CREATE uo\_emp\_data
\end{verbatim}

You can have more than one reference to an object. You might assign an object reference to a variable of the appropriate type, or you might pass an object reference to another object so that it can change or get information from the object.

For more information about object variables and assignment, see “User objects that behave like structures” on page 85.
About objects

Using ancestors and descendants

Descendent objects

In PowerBuilder, an object class can be inherited from another class. The inherited or descendent object has all the instance variables, events, and functions of the ancestor. You can augment the descendant by adding more variables, events, and functions. If you change the ancestor, even after editing the descendant, the descendant incorporates the changes.

Instantiating

When you instantiate a descendent object, PowerBuilder also instantiates all its ancestor classes. You do not have programmatic access to these ancestor instances, except in a few limited ways, such as when you use the scope operator to access an ancestor version of a function or event script.

Garbage collection

What garbage collection does

The PowerBuilder garbage collection mechanism checks memory automatically for unreferenced and orphaned objects and removes any it finds, thus taking care of most memory leaks. You can use garbage collection to destroy objects instead of explicitly destroying them using the DESTROY statement. This lets you avoid runtime errors that occur when you destroy an object that was being used by another process or had been passed by reference to a posted event or function.

When garbage collection occurs

- **When a reference is removed from an object** A reference to an object is any variable whose value is the object. When the variable goes out of scope, or when it is assigned a different value, PowerBuilder removes a reference to the object, counts the remaining references, and destroys the object if no references remain.

- **When the garbage collection interval is exceeded** When PowerBuilder completes the execution of a system-triggered event, it makes a garbage collection pass if the set interval between garbage collection passes has been exceeded. The default interval is 0.5 seconds. The garbage collection pass removes any objects and classes that cannot be referenced, including those containing circular references (otherwise unreferenced objects that reference each other).

When you post an event or function and pass an object reference, PowerBuilder adds an internal reference to the object to prevent it from being collected between the time of the post and the actual execution of the event or function. This reference is removed when the event or function is executed.
<table>
<thead>
<tr>
<th><strong>Exceptions to garbage collection</strong></th>
<th>There are a few objects that are prevented from being collected:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Visual objects</strong></td>
<td>Any object that is visible on your screen is not collected because when the object is created and displayed on your screen, an internal reference is added to the object. When any visual object is closed it is explicitly destroyed.</td>
</tr>
<tr>
<td><strong>Timing objects</strong></td>
<td>Any Timing object that is currently running is not collected because the Start function for a Timing object adds an internal reference. The Stop function removes the reference.</td>
</tr>
<tr>
<td><strong>Shared objects</strong></td>
<td>Registered shared objects are not collected because the SharedObjectRegister function adds an internal reference. SharedObjectUnregister removes the internal reference.</td>
</tr>
</tbody>
</table>

| **Controlling when garbage collection occurs** | Garbage collection occurs automatically in PowerBuilder, but you can use the functions GarbageCollect, GarbageCollectGetTimeLimit, and GarbageCollectSetTimeLimit to force immediate garbage collection or to change the interval between reference count checks. By setting the interval between garbage collection passes to a very large number, you can effectively turn off garbage collection. |

| **User objects that behave like structures** | In PowerBuilder, a nonvisual user object can provide functionality similar to that of a structure. Its instance variables form a collection similar to the variables for the structure. In scripts, you use dot notation to refer to the user object’s instance variables, just as you do for structure variables. |
| **Advantages of user objects** | The user object can include functions and its own structure definitions, and it allows you to inherit from an ancestor class. None of this is possible with a structure definition. |
| **Memory allocation differences** | Memory allocation is different for user objects and structures. An object variable is a reference to the object. Declaring the variable does not allocate memory for the object. After you declare it, you must instantiate it with a CREATE statement. Assignment for a user object is also different (described in "Assignment for objects and structures" next). |
| **Autoinstantiated objects** | If you want a user object that has methods and inheritance but want the memory allocation of a structure, you can define an autoinstantiated object. |
You do not have to create and destroy autoinstantiated objects. Like structures, they are created when they are declared and destroyed when they go out of scope. However, because assignment for autoinstantiated objects behaves like structures, the copies made of the object can be a drawback.

To make a custom class user object autoinstantiated, select the Autoinstantiate check box on the user object’s property sheet.

Assignment for objects and structures

In PowerBuilder, assignment for objects is different from assignment for structures or autoinstantiated objects:

- When you assign one structure to another, the whole structure is copied so that there are two copies of the structure.
- When you assign one object variable to another, the object reference is copied so that both variables point to the same object. There is only one copy of the object.

Events

Assignment for structures

Declaring a structure variable creates an instance of that structure:

```plaintext
str_emp_data str_emp1, str_emp2  // Two structure instances
```

When you assign a structure to another structure, the whole structure is copied and a second copy of the structure data exists:

```plaintext
str_emp1 = str_emp2
```

The assignment copies the whole structure from one structure variable to the other. Each variable is a separate instance of the structure `str_emp_data`.

If the structures have different definitions, you cannot assign one to another, even if they have the same set of variable definitions.
For example, this assignment is not allowed:

```plaintext
str_emp str_person1
str_cust str_person2
str_person2 = str_person1  // Not allowed
```

For information about passing structures as function arguments, see “Passing arguments to functions and events” on page 106.

### Assignment for objects

Declaring an object variable declares an object reference:

```plaintext
uo_emp_data uo_emp1, uo_emp2  // Two object references
```

Using the CREATE statement creates an instance of the object:

```plaintext
uo_emp1 = CREATE uo_emp_data
```

When you assign one object variable to another, a reference to the object instance is copied. Only one copy of the object exists:

```plaintext
uo_emp2 = uo_emp1  // Both point to same object instance
```

Assignments between ancestor and descendent objects occur in the same way, with an object reference being copied to the target object.

Suppose that `uo_emp_data` is an ancestor user object of `uo_emp_active` and `uo_emp_inactive`.

Declare variables of the ancestor type:

```plaintext
uo_emp_data uo_emp1, uo_emp2
```

Create an instance of the descendant and store the reference in the ancestor variable:

```plaintext
uo_emp1 = CREATE USING "uo_emp_active"
```

Assigning `uo_emp1` to `uo_emp2` makes both variables refer to one object that is an instance of the descendant `uo_emp_active`:

```plaintext
uo_emp2 = uo_emp1
```

For information about passing objects as function arguments, see “Passing arguments to functions and events” on page 106.
Assignment for autoinstantiated user objects

Declaring an autoinstantiated user object creates an instance of that object (just like a structure). The CREATE statement is not allowed for objects with the Autoinstantiate setting. In the following example, `uo_emp_data` has the Autoinstantiate setting:

```plaintext
uo_emp_data uo_emp1, uo_emp2 // Two object instances
```

When you assign an autoinstantiated object to another autoinstantiated object, the whole object is copied to the second variable:

```plaintext
uo_emp1 = uo_emp2
```

You never have multiple references to an autoinstantiated user object.

Passing to a function

When you pass an autoinstantiated user object to a function, it behaves like a structure:

- Passing by value passes a copy of the object.
- Passing by reference passes a pointer to the object variable, just as for any standard datatype.
- Passing as read-only passes a copy of the object but that copy cannot be modified.

Restrictions for copying

Assignments are allowed between autoinstantiated user objects only if the object types match or if the target is a nonautoinstantiated ancestor.

**Rule 1** If you assign one autoinstantiated object to another, they must be of the same type.

**Rule 2** If you assign an autoinstantiated descendent object to an ancestor variable, the ancestor cannot have the Autoinstantiate setting. The ancestor variable will contain a reference to a copy of its descendant.

**Rule 3** If you assign an ancestor object to a descendent variable, the ancestor must contain an instance of the descendant or an execution error occurs.

Examples

To illustrate, suppose you have these declarations. `Uo_emp_active` and `uo_emp_inactive` are autoinstantiated objects that are descendants of non-autoinstantiated `uo_emp_data`:

```plaintext
uo_emp_data uo_empl // Ancestor
uo_emp_active uo_empa, uo_empb // Descendants
uo_emp_inactive uo_empi // Another descendant
```
Example of rule 1  When assigning one instance to another from the user objects declared above, some assignments are not allowed by the compiler:

\[
\begin{align*}
\texttt{uo_empb} &= \texttt{uo_empa} \quad \text{// Allowed, same type} \\
\texttt{uo_empa} &= \texttt{uo_empi} \quad \text{// Not allowed, different types}
\end{align*}
\]

Example of rule 2  After this assignment, 
\texttt{uo_emp1} contains a copy of the descendent object 
\texttt{uo_empa}. \texttt{Uo_emp_data} (the type for \texttt{uo_emp1}) must not be autoinstantiated. Otherwise, the assignment violates rule 1. If \texttt{uo_emp1} is autoinstantiated, a compiler error occurs:

\[
\texttt{uo_emp1} = \texttt{uo_empa}
\]

Example of rule 3  This assignment is only allowed if \texttt{uo_emp1} contains an instance of its descendant \texttt{uo_empa}, which it would if the previous assignment had occurred before this one:

\[
\texttt{uo_empa} = \texttt{uo_emp1}
\]

If it did not contain an instance of target descendent type, an execution error would occur.

For more information about passing arguments to functions and events, see “Passing arguments to functions and events” on page 106.
Assignment for objects and structures
C H A P T E R  6

Calling Functions and Events

About this chapter

This chapter provides background information that will help you understand the different ways you can use functions and events. It then provides the syntax for calling functions and events.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About functions and events</td>
<td>91</td>
</tr>
<tr>
<td>Finding and executing functions and events</td>
<td>94</td>
</tr>
<tr>
<td>Triggering versus posting functions and events</td>
<td>96</td>
</tr>
<tr>
<td>Static versus dynamic calls</td>
<td>98</td>
</tr>
<tr>
<td>Overloading, overriding, and extending functions and events</td>
<td>104</td>
</tr>
<tr>
<td>Passing arguments to functions and events</td>
<td>106</td>
</tr>
<tr>
<td>Using return values</td>
<td>109</td>
</tr>
<tr>
<td>Syntax for calling PowerBuilder functions and events</td>
<td>111</td>
</tr>
<tr>
<td>Calling functions and events in an object’s ancestor</td>
<td>114</td>
</tr>
</tbody>
</table>

About functions and events

Importance of functions and events

Much of the power of the PowerScript language resides in the built-in PowerScript functions that you can use in expressions and assignment statements.

Types of functions and events

PowerBuilder objects have built-in events and functions. You can enhance objects with your own user-defined functions and events, and you can declare local external functions for an object. The PowerScript language also has system functions that are not associated with any object. You can define your own global functions and declare external functions and remote procedure calls.
The following table shows the different types of functions and events.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events</td>
<td>Event</td>
<td>An action in an object or control that can start the execution of a script. A user can initiate an event by an action such as clicking an object or entering data, or a statement in another script can initiate the event.</td>
</tr>
<tr>
<td></td>
<td>User event</td>
<td>An event you define to add functionality to an object. You specify the arguments, return value, and whether the event is mapped to a system message. For information about defining user events, see the PowerBuilder <em>Users Guide</em>.</td>
</tr>
<tr>
<td></td>
<td>System or built-in event</td>
<td>An event that is part of an object’s PowerBuilder definition. System events are usually triggered by user actions or system messages. PowerBuilder passes a predefined set of arguments for use in the event’s script. System events either return a long or do not have a return value.</td>
</tr>
<tr>
<td>Functions</td>
<td>Function</td>
<td>A program or routine that performs specific processing.</td>
</tr>
<tr>
<td></td>
<td>System function</td>
<td>A built-in PowerScript function that is not associated with an object.</td>
</tr>
<tr>
<td></td>
<td>Object function</td>
<td>A function that is part of an object’s definition. PowerBuilder has many predefined object functions and you can define your own.</td>
</tr>
<tr>
<td></td>
<td>User-defined function</td>
<td>A function you define. You define global functions in the Function painter and object functions in other painters with Script views.</td>
</tr>
<tr>
<td></td>
<td>Global function</td>
<td>A function you define that can be called from any script. PowerScript’s system functions are globally accessible, but they have a different place in the search order.</td>
</tr>
<tr>
<td></td>
<td>Local external function</td>
<td>An external function that belongs to an object. You declare it in the Window or User Object painter. Its definition is in another library.</td>
</tr>
<tr>
<td></td>
<td>Global external function</td>
<td>An external function that you declare in any painter, making it globally accessible. Its definition is in another library.</td>
</tr>
<tr>
<td></td>
<td>Remote procedure call (RPC)</td>
<td>A stored procedure in a database that you can call from a script. The declaration for an RPC can be global or local (belonging to an object). The definition for the procedure is in the database.</td>
</tr>
</tbody>
</table>
Comparing functions and events

Functions and events have the following similarities:

- Both functions and events have arguments and return values.
- You can call object functions and events dynamically or statically. Global or system functions cannot be called dynamically.
- You can post or trigger a function or event call.

Functions and events have the following differences:

- Functions can be global or part of an object’s definition. Events are associated only with objects.
- PowerBuilder uses different search orders when looking for events and functions.
- A call to an undefined function triggers an error. A call to an undefined event does not trigger an error.
- Object-level functions can be overloaded. Events (and global functions) cannot be overloaded.
- When you define a function, you can restrict access to it. You cannot add scope restrictions when you define events.
- When functions are inherited, you can extend the ancestor function by calling it in the descendant’s script. You can also override the function definition. When events are inherited, the scripts for those events are extended by default. You can choose to extend or override the script.

Which to use

Whether you write most of your code in user-defined functions or in event scripts is one of the design decisions you must make. Because there is no performance difference, the decision is based on how you prefer to interact with PowerBuilder: whether you prefer the interface for defining user events or that for defining functions, how you want to handle errors, and whether your design includes overloading.

It is unlikely that you will use either events or functions exclusively, but for ease of maintenance, you might want to choose one approach for handling most situations.
Finding and executing functions and events

PowerBuilder looks for a matching function or event based on its name and its argument list. PowerBuilder can make a match between compatible datatypes (such as all the numeric types). The match does not have to be exact. PowerBuilder ranks compatible datatypes to quantify how closely one datatype matches another.

A major difference between functions and events is how PowerBuilder looks for them.

Finding functions

When calling a function, PowerBuilder searches until it finds a matching function and executes it—the search ends. Using functions with the same name but different arguments is called function overloading. For more information, see “Overloading, overriding, and extending functions and events” on page 104.

If you do not qualify a function name with an object, PowerBuilder searches for the function and executes the first one it finds that matches the name and arguments. It searches for a match in the following order:

1. A global external function.
2. A global function.
3. An object function and local external function. If the object is a descendant, PowerBuilder searches upward through the ancestor hierarchy to find a match for the function prototype.
4. A system function.

DataWindow expression functions

The functions that you use in the DataWindow painter in expressions for computed fields, filters, validation rules, and graphed data cannot be overridden. For example, if you create a global function called Today, it is used instead of the PowerScript system function Today, but it is not used instead of the DataWindow expression function Today.
Qualified function names

You can qualify an object function using dot notation to ensure that the object function is found, not a global function of the same name. With a qualified name, the search for a matching function involves the ancestor hierarchy only (item 3 in the search list above), as shown in the following examples of function calls:

```
dw_1.Update(  )
w_employee.uf_process_list()
This.uf_process_list()
```

When PowerBuilder searches the ancestor hierarchy for a function, you can specify that you want to call an ancestor function instead of a matching descendant function.

For the syntax for calling ancestor functions, see “Calling functions and events in an object’s ancestor” on page 114.

Finding events

PowerBuilder events in descendent objects are, by default, extensions of ancestor events. PowerBuilder searches for events in the object’s ancestor hierarchy until it gets to the top ancestor or finds an event that overrides its ancestor. Then it begins executing the events, from the ancestor event down to the descendent event.

The following illustration shows the difference between searching for events and searching for functions:

![Finding functions versus events diagram](attachment:image.png)
In PowerBuilder, when you trigger a function or event, it is called immediately. Its return value is available for use in the script.

When you post a function or event, it is added to the object’s queue and executed in its turn. In most cases, it is executed when the current script is finished; however, if other system events have occurred in the meantime, its position in the queue might be after other scripts. Its return value is not available to the calling script.

Because POST makes the return value unavailable to the caller, you can think of it as turning the function or event call into a statement.

Use posting when activities need to be finished before the code checks state information or does further processing (see Example 2 below).

All events posted by PowerBuilder are processed by a separate queue from the Windows system queue. PowerBuilder posted messages are processed before Windows posted messages, so PowerBuilder events that are posted in an event that posts a Windows message are processed before the Windows message.

For example, when a character is typed into an EditMask control, the PowerBuilder pdm_keydown event posts the Windows message WM_CHAR to enter the character. If you want to copy the characters as they are entered from the EditMask control to another control, do not place the code in an event posted in the pdm_keydown event. The processing must take place in an event that occurs after the WM_CHAR message is processed, such as in an event mapped to pdm_keyup.

Because no value is returned, you:

- Cannot use a posted function or event as an operand in an expression
- Cannot use a posted function or event as the argument for another function
- Can only use POST on the last call in a cascaded sequence of calls

These statements cause a compiler error. Both uses require a return value:

```
IF POST IsNull( ) THEN ...
w_1.uf_getresult(dw_1.POST GetBorderStyle(2))
```
Asynchronous processing in EAServer

Using POST is not supported in the context of calls to EAServer components. For how to simulate asynchronous processing by posting a call to a shared object on an EAServer client, see the SharedObjectGet function in the online Help. For information about asynchronous processing in EAServer, see the EAServer documentation for the ThreadManager and MessageService modules.

**TriggerEvent and PostEvent functions**

For backward compatibility, the TriggerEvent and PostEvent functions are still available, but you cannot pass arguments to the called event. You must pass data to the event in PowerBuilder’s Message object.

Examples of posting

The following examples illustrate how to post events.

**Example 1**  In a sample application, the Open event of the `w_activity_manager` window calls the functions `uf_setup` and `uf_set_tabpgsystem`. (The functions belong to the user object `u_app_actman`.) Because the functions are posted, the Open event is allowed to finish before the functions are called. The result is that the window is visible while setup processing takes place, giving the user something to look at:

```powerbuilder
  guo_global_vars.iuo_app_actman.POST uf_setup()
  guo_global_vars.iuo_com_actman.POST
  uf_set_tabpgsystem(0)
```

**Example 2**  In a sample application, the DoubleClicked event of the `tv_roadmap` TreeView control in the `u_tabpg_amroadmap` user object posts a function that processes the TreeView item. If the event is not posted, the code that checks whether to change the item’s picture runs before the item’s expanded flag is set:

```powerbuilder
  parent.POST uf_process_item ()
```
Static versus dynamic calls

PowerBuilder calls functions and events in three ways, depending on the type of function or event and the lookup method defined.

Table 6-2: How PowerBuilder calls functions and events

<table>
<thead>
<tr>
<th>Type of function</th>
<th>Compiler typing</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global and system functions</td>
<td>Strongly typed. The function must exist when the script is compiled.</td>
<td>These functions must exist and are called directly. They are not polymorphic, and no substitution is ever made at execution time.</td>
</tr>
<tr>
<td>Object functions with STATIC lookup</td>
<td>Strongly typed. The function must exist when the script is compiled.</td>
<td>The functions are polymorphic. They must exist when you compile, but if another class is instantiated at execution time, its function is called instead.</td>
</tr>
<tr>
<td>Object functions with DYNAMIC lookup</td>
<td>Weakly typed. The function does not have to exist when the script is compiled.</td>
<td>The functions are polymorphic. The actual function called is determined at execution time.</td>
</tr>
</tbody>
</table>

Specifying static or dynamic lookup

For object functions and events, you can choose when PowerBuilder looks for them by specifying static or dynamic lookup. You specify static or dynamic lookup using the STATIC or DYNAMIC keywords. The DYNAMIC keyword applies only to functions that are associated with an object. You cannot call global or system functions dynamically.

Static calls

By default, PowerBuilder makes static lookups for functions and events. This means that it identifies the function or event by matching the name and argument types when it compiles the code. A matching function or event must exist in the object at compile time.

Results of static calls

Static calls do not guarantee that the function or event identified at compile time is the one that is executed. Suppose that you define a variable of an ancestor type and it has a particular function definition. If you assign an instance of a descendent object to the variable and the descendent has a function that overrides the ancestor’s function (the one found at compile time), the function in the descendant is executed.
Dynamic calls

When you specify a dynamic call in PowerBuilder, the function or event does not have to exist when you compile the code. You are indicating to the compiler that there will be a suitable function or event available at execution time.

For a dynamic call, PowerBuilder waits until it is time to execute the function or event to look for it. This gives you flexibility and allows you to call functions or events in descendants that do not exist in the ancestor.

To illustrate the results of dynamic calls, consider these objects:

- Ancestor window w_a with a function Set(integer).
- Descendent window w_a_desc with two functions: Set(integer) overrides the ancestor function, and Set(string) is an overload of the function.

**Situation 1** Suppose you open the window mywindow of the ancestor window class w_a:

```powerbuilder
w_a mywindow
Open(mywindow)
```

This is what happens when you call the Set function statically or dynamically:

<table>
<thead>
<tr>
<th>This statement</th>
<th>Has this result</th>
</tr>
</thead>
<tbody>
<tr>
<td>mywindow.Set(1)</td>
<td>Compiles correctly because function is found in the ancestor w_a. At runtime, Set(integer) in the ancestor is executed.</td>
</tr>
<tr>
<td>mywindow.Set(&quot;hello&quot;)</td>
<td>Fails to compile; no function prototype in w_a matches the call.</td>
</tr>
<tr>
<td>mywindow.DYNAMIC Set(&quot;hello&quot;)</td>
<td>Compiles successfully because of the DYNAMIC keyword. An error occurs at runtime because no matching function is found.</td>
</tr>
</tbody>
</table>

**Situation 2** Now suppose you open mywindow as the descendant window class w_a_desc:

```powerbuilder
w_a mywindow
Open(mywindow, "w_a_desc")
```
Static versus dynamic calls

This is what happens when you call the Set function statically or dynamically in the descendant window class:

<table>
<thead>
<tr>
<th>This statement</th>
<th>Has this result</th>
</tr>
</thead>
<tbody>
<tr>
<td>mywindow.Set(1)</td>
<td>Compiles correctly because function is found in the ancestor w_a. At runtime, Set(integer) in the descendant is executed.</td>
</tr>
<tr>
<td>mywindow.Set(&quot;hello&quot;)</td>
<td>Fails to compile; no function prototype in the ancestor matches the call.</td>
</tr>
<tr>
<td>mywindow.DYNAMIC Set(&quot;hello&quot;)</td>
<td>Compiles successfully because of the DYNAMIC keyword. At runtime, Set(string) in the descendant is executed.</td>
</tr>
</tbody>
</table>

Disadvantages of dynamic calls

- **Slower performance**  Because dynamic calls are resolved at runtime, they are slower than static calls. If you need the fastest performance, design your application to avoid dynamic calls.

- **Less error checking** When you use dynamic calls, you are foregoing error checking provided by the compiler. Your application is more open to application errors, because functions that are called dynamically might be unavailable at execution time. Do not use a dynamic call when a static call will suffice.

Example using dynamic call

A sample application has an ancestor window w_datareview_frame that defines several functions called by the menu items of m_datareview_framemenu. They are empty stubs with empty scripts so that static calls to the functions will compile. Other windows that are descendants of w_datareview_frame have scripts for these functions, overriding the ancestor version.

The wf_print function is one of these—it has an empty script in the ancestor and appropriate code in each descendent window:

```
gruo_global_vars.ish_currentsheet.wf_print ()
```

The wf_export function called by the m_export item on the m_file menu does not have a stubbed-out version in the ancestor window. This code for m_export uses the DYNAMIC keyword to call wf_export. When the program runs, the value of variable ish_currentsheet is a descendent window that does have a definition for wf_export:

```
gruo_global_vars.ish_currentsheet.DYNAMIC wf_export ()
```
Errors when calling functions and events dynamically

If you call a function or event dynamically, different conditions create different results, from no effect to an execution error. The tables in this section illustrate this.

Functions

The rules for functions are similar to those for events, except functions must exist: if a function is not found, an error always occurs. Although events can exist without a script, if a function is defined it has to have code. Consider the following statements:

1. This statement calls a function without looking for a return value:
   ```
   object.DYNAMIC funcname( )
   ```

2. This statement looks for an integer return value:
   ```
   int li_int
   li_int = object.DYNAMIC funcname( )
   ```

3. This statement looks for an Any return value:
   ```
   any la_any
   la_any = object.DYNAMIC funcname( )
   ```

The following table uses these statements as examples.

<table>
<thead>
<tr>
<th>Condition 1</th>
<th>Condition 2</th>
<th>Result</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>The function does not exist.</td>
<td>None.</td>
<td>Execution error 65: Dynamic function not found.</td>
<td>All the statements cause error 65.</td>
</tr>
<tr>
<td>The function is found and executed but is not defined with a return value.</td>
<td>The code is looking for a return value.</td>
<td>Execution error 63: Function/event with no return value used in expression.</td>
<td>Statements 2 and 3 cause error 63.</td>
</tr>
</tbody>
</table>

Events

Consider these statements:

1. This statement calls an event without looking for a return value:
   ```
   object.EVENT DYNAMIC eventname( )
   ```

2. This example looks for an integer return value:
   ```
   int li_int
   li_int = object.EVENT DYNAMIC eventname( )
   ```
This example looks for an Any return value:

```powershell
any la_any
la_any = object.EVENT DYNAMIC eventname()
```

The following table uses these statements as examples.

**Table 6-4: Dynamic event calling errors**

<table>
<thead>
<tr>
<th>Condition 1</th>
<th>Condition 2</th>
<th>Result</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>The event does not exist.</td>
<td>The code is not looking for a return value.</td>
<td>Nothing; the call fails silently.</td>
<td>Statement 1 fails but does not cause an error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A null of the Any datatype is returned.</td>
<td><em>La_any</em> is set to null in statement 3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the expected datatype is not Any, execution error 19 occurs: Cannot convert Any in Any variable to datatype.</td>
<td>The assignment to <em>li_int</em> causes execution error 19 in statement 2.</td>
</tr>
<tr>
<td>The event is found but is not implemented (there is no script).</td>
<td>The event has a defined return value.</td>
<td>A null of the defined datatype is returned.</td>
<td>If eventname is defined to return integer, <em>li_int</em> is set to null in statement 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A null of the Any datatype is returned.</td>
<td><em>La_any</em> is set to null in statement 3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the expected datatype is not Any, execution error 19 occurs: Cannot convert Any in Any variable to datatype.</td>
<td>The assignment to <em>li_int</em> causes execution error 19 in statement 2.</td>
</tr>
<tr>
<td>The event is found and executed but is not defined with a return value.</td>
<td>The code is looking for a return value.</td>
<td>Execution error 63: Function/event with no return value used in expression.</td>
<td>Statements 2 and 3 cause error 63.</td>
</tr>
</tbody>
</table>
When an error occurs

You can surround a dynamic function call in a try-catch block to prevent the application from terminating when an execution error occurs. Although you can also handle the error in the SystemError event, you should not allow the application to continue once the SystemError event is invoked—the SystemError event should only clean up and halt the application.

For information on using try-catch blocks, see the chapter on exception handling in Application Techniques.

If the arguments do not match

Function arguments are part of the function’s definition. Therefore, if the arguments do not match (a compatible match, not an exact match), it is essentially a different function. The result is the same as if the function did not exist.

If you call an event dynamically and the arguments do not match, the call fails and control returns to the calling script. There is no error.

Error-proofing your code

Calling functions and events dynamically opens up your application to potential errors. The surest way to avoid these errors is to always make static calls to functions and events. When that is not possible, your design and testing can ensure that there is always an appropriate function or event with the correct return datatype.

One type of error you can check for and avoid is data conversion errors.

The preceding tables illustrated that a function or event can return a null value either as an Any variable or as a variable of the expected datatype when a function or event definition exists but is not implemented.

If you always assign return values to Any variables for dynamic calls, you can test for null (which indicates failure) before using the value in code.

This example illustrates the technique of checking for null before using the return value.

```powershell
any la_any
integer li_gotvalue
la_any = object.DYNAMIC uf_getaninteger() IF IsNull(la_any) THEN
    ... // Error handling
ELSE
    li_gotvalue = la_any
END IF
```
Overloading, overriding, and extending functions and events

In PowerBuilder, when functions are inherited, you can choose to overload or override the function definition, described in “Overloading and overriding functions” next.

When events are inherited, the scripts for those events are extended by default. You can choose to extend or override the script, described in “Extending and overriding events” on page 105.

Overloading and overriding functions

To create an overloaded function, you declare the function as you would any function using Insert>Function.

Overriding means defining a function in a descendent object that has the same name and argument list as a function in the ancestor object. In the descendent object, the function in the descendant is always called instead of the one in the ancestor—unless you use the scope resolution operator (::).

To override a function, open the descendent object in the painter, select the function in the Script view, and code the new script. The icon that indicates that there is a script for a function is half shaded when the function is inherited from an ancestor.

You can overload or override object functions only—you cannot overload global functions.

Type promotion when matching arguments for overloaded functions

When you have overloaded a function so that one version handles numeric values and another version handles strings, it is clear to the programmer what arguments to provide to call each version of the function. Overloading with unrelated datatypes is a good idea and can provide needed functionality for your application.

Problematic overloading

If different versions of a function have arguments of related datatypes (different numeric types or strings and chars), you must consider how PowerBuilder promotes datatypes in determining which function is called. This kind of overloading is undesirable because of potential confusion in determining which function is called.
When you call a function with an expression as an argument, the datatype of the expression might not be obvious. However, the datatype is important in determining what version of an overloaded function is called.

Because of the intricacies of type promotion for numeric datatypes, you might decide that you should not define overloaded functions with different numeric datatypes. Changes someone makes later can affect the application more drastically than expected if the change causes a different function to be called.

How type promotion works

When PowerBuilder evaluates an expression, it converts the datatypes of constants and variables so that it can process or combine them correctly.

Numbers When PowerBuilder evaluates numeric expressions, it promotes the datatypes of values according to the operators and the datatypes of the other operands. For example, the datatype of the expression n/2 is double because it involves division—the datatype of n does not matter.

Strings When evaluating an expression that involves chars and strings, PowerBuilder promotes chars to strings.

For more information on type promotion, see “Datatype of PowerBuilder expressions” on page 75.

Using conversion functions

You can take control over the datatypes of expressions by calling a conversion function. The conversion function ensures that the datatype of the expression matches the function prototype you want to call.

For example, because the expression n/2 involves division, the datatype is double. However, if the function you want to call expects a long, you can use the Long function to ensure that the function call matches the prototype:

\[
\text{CalculateHalf}(\text{Long}(n/2))
\]

Extending and overriding events

In PowerBuilder, when you write event scripts in a descendent object, you can extend or override scripts that have been written in the ancestor.

Extending (the default) means executing the ancestor’s script first, then executing code in the descendant’s event script.

Overriding means ignoring the ancestor’s script and only executing the script in the descendent.
No overloaded events
You cannot overload an event by defining an event with the same name but different arguments. Event names must be unique.

To select extending or overriding, open the script in the Script view and check or clear the Extend Ancestor Script item in the Edit or pop-up menu.

Passing arguments to functions and events

In PowerBuilder, arguments for built-in or user-defined functions and events can be passed three ways:

<table>
<thead>
<tr>
<th>Method of passing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>By value</td>
<td>A copy of the variable is available in the function or event script. Any changes to its value affect the copy only. The original variable in the calling script is not affected.</td>
</tr>
<tr>
<td>By reference</td>
<td>A pointer to the variable is passed to the function or event script. Changes affect the original variable in the calling script.</td>
</tr>
<tr>
<td>Read-only</td>
<td>The variable is available in the function or event. Its value is treated as a constant—changes to the variable are not allowed and cause a compiler error. Read-only provides a performance advantage for some datatypes because it does not create a copy of the data, as with by value. Datatypes for which read-only provides a performance advantage are string, blob, date, time, and DateTime. For other datatypes, read-only provides documentation for other developers by indicating something about the purpose of the argument.</td>
</tr>
</tbody>
</table>

Passing objects

When you pass an object to a function or event, the object must exist when you refer to its properties and functions. If you call the function but the object has been destroyed, you get the execution error for a null object reference. This is true whether you pass by reference, by value, or read-only.
To illustrate, suppose you have a window with a SingleLineEdit. If you post a function in the window’s Close event and pass the SingleLineEdit, the object does not exist when the function executes. To use information from the SingleLineEdit, you must pass the information itself, such as the object’s text, rather than the object.

When passing an object, you never get another copy of the object. By reference and by value affect the object reference, not the object itself.

**Objects passed by value**

When you pass an object by value, you pass a copy of the reference to the object. That reference is still pointing to the original object. If you change properties of the object, you are changing the original object. However, you can change the value of the variable so that it points to another object without affecting the original variable.

**Objects passed by reference**

When you pass an object by reference, you pass a pointer to the original reference to the object. Again, if you change properties of the object, you are changing the original object. You can change the value of the variable that was passed, but the result is different—the original reference now points to the new object.

**Objects passed as read-only**

When you pass an object as read-only, you get a copy of the reference to the object. You cannot change the reference to point to a new object (because read-only is equivalent to a CONSTANT declaration), but you can change properties of the object.

### Passing structures

Structures as arguments behave like simple variables, not like objects.

**Structures passed by value**

When you pass a structure by value, PowerBuilder passes a copy of the structure. You can modify the copy without affecting the original.

**Structures passed by reference**

When you pass a structure by reference, PowerBuilder passes a reference to the structure. When you changes values in the structure, you are modifying the original. You will not get a null object reference, because structures always exist until they go out of scope.

**Structures passed as read-only**

When you pass a structure as read-only, PowerBuilder passes a copy of the structure. You cannot modify any members of the structure.
Passing arrays

When an argument is an array, you specify brackets as part of the argument name in the declaration for the function or event.

For example, suppose a function named `uf_convertarray` accepts a variable-size array of integers. If the argument’s name is `intarray`, then for Name enter `intarray[]` and for Type enter `integer`.

In the script that calls the function, you either declare an array variable or use an instance variable or value that has been passed to you. The declaration of that variable, wherever it is, looks like this:

```powershell
integer a[]
```

When you call the function, omit the brackets, because you are passing the whole array. If you specified brackets, you would be passing one value from the array:

```powershell
uf_convertarray(a)
```

For comparison, suppose the `uf_convertarray` function accepts a fixed-size array of integers of 10 elements instead. If the argument’s name is `intarray`, then for Name enter `intarray[10]`, and for Type enter `integer`.

The declaration of the variable to be passed looks like this:

```powershell
integer a[10]
```

You call the function the same way, without brackets:

```powershell
uf_convertarray(a)
```

If the array dimensions do not match

If the dimensions of the array variable passed do not match the dimensions declared for the array argument, then array-to-array assignment rules apply. For more information, see “Declaring arrays” on page 48.
CHAPTER 6 Calling Functions and Events

Using return values

You can use return values of functions and events.

Functions

All built-in PowerScript functions return a value. You can use the return value or ignore it. User-defined functions and external functions might or might not return a value.

To use a return value, assign it to a variable of the appropriate datatype or call the function wherever you can use a value of that datatype.

Posting a function

If you post a function, you cannot use its return value.

Examples

The built-in Asc function takes a string as an argument and returns the Unicode code point value of the string’s first character:

```power_script
string S1 = "Carton"
long Test
Test=32+Asc(S1)  // Test now contains the value 99
    // (the code point value of "C" is 67).
```

The SelectRow function expects a row number as the first argument. The return value of the GetRow function supplies the row number:

```power_script
dw_1.SelectRow(dw_1.GetRow(), true)
```

To ignore a return value, call the function as a single statement:

```power_script
Beep(4)  // This returns a value, but it is
    // rarely needed.
```

Events

Most system events return a value. The return value is a long—numeric codes have specific meanings for each event. You specify the event’s return code with a RETURN statement in the event script.

When the event is triggered by user actions or system messages, the value is returned to the system, not to a script you write.
Using return values

When you trigger a system or user-defined event, the return value is returned to your script and you can use the value as appropriate. If you post an event, you cannot use its return value.

Using cascaded calling and return values

PowerBuilder dot notation allows you to chain together several object function or event calls. The return value of the function or event becomes the object for the following call.

This syntax shows the relationship between the return values of three cascaded function calls:

```
func1returnsobject( ).func2returnsobject( ).func3returnsanything( )
```

Disadvantage of cascaded calls

When you call several functions in a cascade, you cannot check their return values and make sure they succeeded. If you want to check return values (and checking is always a good idea), call each function separately and assign the return values to variables. Then you can use the verified variables in dot notation before the final function name.

Dynamic calls

If you use the `DYNAMIC` keyword in a chain of cascaded calls, it carries over to all function calls that follow.

In this example, both `func1` and `func2` are called dynamically:

```
object1.DYNAMIC func1().func2()
```

The compiler reports an error if you use `DYNAMIC` more than once in a cascaded call. This example would cause an error:

```
object1.DYNAMIC func1().DYNAMIC func2() // error
```

Posted functions and events

Posted functions and events do not return a value to the calling scripts. Therefore, you can only use `POST` for the last function or event in a cascaded call. Calls before the last must return a valid object that can be used by the following call.

System events

System events can only be last in a cascaded list of calls, because their return value is a long (or they have no return value). They do not return an object that can be used by the next call.

An event you have defined can have a return value whose datatype is an object. You can include such events in a cascaded call.
Syntax for calling PowerBuilder functions and events

Description
This syntax is used to call all PowerBuilder functions and events. Depending on the keywords used, this syntax can be used to call system, global, object, user-defined, and external functions as well as system and user-defined events.

Syntax

{ objectname } { type } { calltype } { when } name ( { argumentlist } )

The following table describes the arguments used in function and event calls.

Table 6-6: Arguments for calling functions and events

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectname (optional)</td>
<td>The name of the object where the function or event is defined followed by a period or the descendant of that object/the name of the ancestor class followed by two colons. If a function name is not qualified, PowerBuilder uses the rules for finding functions and executes the first matching function it finds. For system or global functions, omit objectname. For the rules PowerBuilder uses to find unqualified function names, see “Finding and executing functions and events” on page 94.</td>
</tr>
<tr>
<td>type (optional)</td>
<td>A keyword specifying whether you are calling a function or event. Values are: • FUNCTION (Default) • EVENT</td>
</tr>
<tr>
<td>calltype (optional)</td>
<td>A keyword specifying when PowerBuilder looks for the function or event. Values are: • STATIC (Default) • DYNAMIC For more information about static versus dynamic calls, see “Static versus dynamic calls” on page 98. For more information on dynamic calls, see “Dynamic calls” on page 99.</td>
</tr>
<tr>
<td>when (optional)</td>
<td>A keyword specifying whether the function or event should execute immediately or after the current script is finished. Values are: • TRIGGER – (Default) Execute it immediately. • POST – Put it in the object’s queue and execute it in its turn, after other pending messages have been handled. For more about triggering and posting, see “Triggering versus posting functions and events” on page 96.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the function or event you want to call.</td>
</tr>
<tr>
<td>argumentlist (optional)</td>
<td>The values you want to pass to name. Each value in the list must have a datatype that corresponds to the declared datatype in the function or event definition or declaration.</td>
</tr>
</tbody>
</table>
Syntax for calling PowerBuilder functions and events

Usage

Function and event names are not case sensitive. For example, the following three statements are equivalent:

- `Clipboard("PowerBuilder")`
- `clipboard("PowerBuilder")`
- `CLIPBOARD("PowerBuilder")`

**Calling arguments**
The type, calltype, and when keywords can be in any order after `objectname`.

Not all options in the syntax apply to all types. For example, there is no point in calling a system PowerScript object function dynamically. It always exists, and the dynamic call incurs extra overhead. However, if you had a user-defined function of the same name that applied to a different object, you might call that function dynamically.

User-defined global functions and system functions can be triggered or posted but they cannot be called dynamically.

**Finding functions**
If a global function does not exist with the given name, PowerBuilder will look for an object function that matches the name and argument list before it looks for a PowerBuilder system function.

**Calling functions and events in the ancestor**
If you want to circumvent the usual search order and force PowerBuilder to find a function or event in an ancestor object, bypassing it in the descendant, use the ancestor operator (`::`).

For more information about the scope operator for ancestors, see “Calling functions and events in an object’s ancestor” on page 114.

**Cascaded calls**
Calls can be cascaded using dot notation. Each function or event call must return an object type that is the appropriate object for the following call.

For more information about cascaded calls, see “Using cascaded calling and return values” on page 110.

**Using return values**
If the function has a return value, you can call the function on the right side of an assignment statement, as an argument for another function, or as an operand in an expression.

**External functions**
Before you can call an external function, you must declare it. For information about declaring external functions, see “Declaring external functions” on page 57.
Examples

Example 1  The following statements show various function calls using the most simple construction of the function call syntax.

This statement calls the system function Asc:

```
charnum = Asc("x")
```

This statement calls the DataWindow function in a script that belongs to the DataWindow:

```
Update()
```

This statement calls the global user-defined function gf_setup_appl:

```
gf_setup_appl(24, "Window1")
```

This statement calls the system function PrintRect:

```
PrintRect(job, 250, 250, 7500, 1000, 50)
```

Example 2  The following statements show calls to global and system functions.

This statement posts the global user-defined function gf_setup_appl. The function is executed when the calling script finishes:

```
POST gf_setup_appl(24, "Window1")
```

This statement posts the system function PrintRect. It is executed when the calling script finishes. The print job specified in job must still be open:

```
POST PrintRect(job, 250, 250, 7500, 1000, 50)
```

Example 3  In a script for a control, these statements call a user-defined function defined in the parent window. The statements are equivalent, because FUNCTION, STATIC, and TRIGGER are the defaults:

```
Parent.FUNCTION STATIC TRIGGER wf_process()
Parent.wf_process()
```

Example 4  This statement in a DataWindow control’s Clicked script calls the DoubleClicked event for the same control. The arguments the system passed to Clicked are passed on to DoubleClicked. When triggered by the system, PowerBuilder passes DoubleClicked those same arguments:

```
This.EVENT DoubleClicked(xpos, ypos, row, dwo)
```

This statement posts the same event:

```
This.EVENT POST DoubleClicked(xpos, ypos, row, dwo)
```
Example 5  The variable \textit{iw\_a} is an instance variable of an ancestor window type \textit{w\_ancestorsheet}:

\[
w\_ancestorsheet\ iw\_a
\]

A menu has a script that calls the \textit{wf\_export} function, but that function is not defined in the ancestor. The \textsc{DYNAMIC} keyword is required so that the script compiles:

\[
iw\_a.DYNAMIC\ wf\_export( )
\]

At execution time, the window that is opened is a descendant with a definition of \textit{wf\_export}. That window is assigned to the variable \textit{iw\_a} and the call to \textit{wf\_export} succeeds.

\section*{Calling functions and events in an object’s ancestor}

\textbf{Description} \hfill In PowerBuilder, when an object is instantiated with a descendant object, even if its class is the ancestor and that descendant has a function or event script that overrides the ancestor’s, the descendant’s version is the one that is executed. If you specifically want to execute the ancestor’s version of a function or event, you can use the ancestor operator ($::$) to call the ancestor’s version explicitly.

\textbf{Syntax} \hfill \{ \texttt{objectname, } \texttt{ancestorclass} :: \{ \texttt{type} \} \{ \texttt{when} \} \texttt{name} ( \{ \texttt{argumentlist} \} ) \}

The following table describes the arguments used to call functions and events in an object’s ancestor.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Argument} & \textbf{Description} \\
\hline
\texttt{objectname} & The name of the object whose ancestor contains the function you want to execute. \\
\texttt{(optional)} & \\
\hline
\texttt{ancestorclass} & The name of the ancestor class whose function or event you want to execute. The pronoun \texttt{Super} provides the appropriate reference when \texttt{ancestorobject} is the immediate ancestor of the current object. \\
\hline
\texttt{type} & A keyword specifying whether you are calling a function or event. Values are: \\
\texttt{(optional)} & \\
\hline
\texttt{FUNCTION} & \texttt{EVENT} \\
\hline
\end{tabular}
\end{table}
Usage

The AncestorReturnValue variable When you extend an event script in a descendant object, the compiler automatically generates a local variable called AncestorReturnValue that you can use if you need to know the return value of the ancestor event script. The variable is also generated if you override the ancestor script and use the CALL syntax to call the ancestor event script.

The datatype of the AncestorReturnValue variable is always the same as the datatype defined for the return value of the event. The arguments passed to the call come from the arguments that are passed to the event in the descendant object.

Extending event scripts The AncestorReturnValue variable is always available in extended event scripts. When you extend an event script, PowerBuilder generates the following syntax and inserts it at the beginning of the event script:

```
CALL SUPER::event_name
```

You only see the statement if you export the syntax of the object or look at it in the Source editor.

The following example illustrates the code you can put in an extended event script:

```
If AncestorReturnValue = 1 THEN
  // execute some code
ELSE
  // execute some other code
END IF
```

Overriding event scripts The AncestorReturnValue variable is only available when you override an event script after you call the ancestor event using either of these versions of the CALL syntax:

```
CALL SUPER::event_name
CALL ancestor_name::event_name
```
The compiler cannot differentiate between the keyword SUPER and the name of the ancestor. The keyword is replaced with the name of the ancestor before the script is compiled.

The AncestorReturnValue variable is only declared and a value assigned when you use the CALL event syntax. It is not declared if you use the new event syntax:

\[
\text{ancestor\_name::EVENT event\_name( )}
\]

You can use the same code in a script that overrides its ancestor event script, but you must insert a CALL statement before you use the AncestorReturnValue variable.

\[
\begin{align*}
&\text{// execute code that does some preliminary processing} \\
&\text{CALL SUPER::uo\_myevent} \\
&\text{IF AncestorReturnValue = 1 THEN} \\
&\quad \ldots
\end{align*}
\]

For information about CALL, see CALL on page 123.

**Examples**

**Example 1**  Suppose a window \texttt{w\_ancestor} has an event \texttt{ue\_process}. A descendent window has a script for the same event.

This statement in a script in the descendant searches the event chain and calls all appropriate events. If the descendant extends the ancestor script, it calls a script for each ancestor in turn followed by the descendent script. If the descendant overrides the ancestor, it calls the descendent script only:

\[
\text{EVENT ue\_process( )}
\]

This statement calls the ancestor event only (this script works if the calling script belongs to another object or the descendent window):

\[
\text{w\_ancestor::EVENT ue\_process( )}
\]

**Example 2**  You can use the pronoun \texttt{Super} to refer to the ancestor. This statement in a descendent window script or in a script for a control on that window calls the Clicked script in the immediate ancestor of that window.

\[
\text{Super::EVENT Clicked(0, x, y)}
\]

**Example 3**  These statements call a function \texttt{wf\_myfunc} in the ancestor window (presumably, the descendent also has a function called \texttt{wf\_myfunc}):

\[
\begin{align*}
&\text{Super::wf\_myfunc( )} \\
&\text{Super::POST wf\_myfunc( )}
\end{align*}
\]
PART 2

Statements, Events, and Functions
CHAPTER 7

PowerScript Statements

About this chapter
This chapter describes the PowerScript statements and how to use them in scripts.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>120</td>
</tr>
<tr>
<td>CALL</td>
<td>123</td>
</tr>
<tr>
<td>CHOOSE CASE</td>
<td>124</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>126</td>
</tr>
<tr>
<td>CREATE</td>
<td>127</td>
</tr>
<tr>
<td>DESTROY</td>
<td>130</td>
</tr>
<tr>
<td>DO...LOOP</td>
<td>131</td>
</tr>
<tr>
<td>EXIT</td>
<td>133</td>
</tr>
<tr>
<td>FOR...NEXT</td>
<td>134</td>
</tr>
<tr>
<td>GOTO</td>
<td>136</td>
</tr>
<tr>
<td>HALT</td>
<td>137</td>
</tr>
<tr>
<td>IF...THEN</td>
<td>138</td>
</tr>
<tr>
<td>RETURN</td>
<td>140</td>
</tr>
<tr>
<td>THROW</td>
<td>141</td>
</tr>
<tr>
<td>THROWS</td>
<td>142</td>
</tr>
<tr>
<td>TRY...CATCH...FINALLY...END TRY</td>
<td>143</td>
</tr>
</tbody>
</table>
Assignment

Description
Assigns values to variables or object properties or object references to object variables.

Syntax
variablename = expression

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variablename</td>
<td>The name of the variable or object property to which you want to assign a value. Variablename can include dot notation to qualify the variable with one or more object names.</td>
</tr>
<tr>
<td>expression</td>
<td>An expression whose datatype is compatible with variablename.</td>
</tr>
</tbody>
</table>

Usage
Use assignment statements to assign values to variables. To assign a value to a variable anywhere in a script, use the equal sign (=). For example:

```plaintext
String1 = "Part is out of stock"
TaxRate = .05
```

**No multiple assignments** Since the equal sign is also a logical operator, you cannot assign more than one variable in a single statement. For example, the following statement does not assign the value 0 to A and B:

```plaintext
A=B=0      // This will not assign 0 to A and B.
```

This statement first evaluates B=0 to true or FALSE and then tries to assign this boolean value to A. When A is not a boolean variable, this line produces an error when compiled.

**Assigning array values** You can assign multiple array values with one statement, such as:

```plaintext
int Arr[]
Arr = {1, 2, 3, 4}
```

You can also copy array contents. For example, this statement copies the contents of Arr2 into array Arr1:

```plaintext
Arr1 = Arr2
```
Operator shortcuts  The PowerScript shortcuts for assigning values to variables in the following table have slight performance advantages over their equivalents.

Table 7-1: Shortcuts for assigning values

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Example</th>
<th>Equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>++ i</td>
<td>i = i + 1</td>
</tr>
<tr>
<td>--</td>
<td>-- i</td>
<td>i = i - 1</td>
</tr>
<tr>
<td>+=</td>
<td>i += 3</td>
<td>i = i + 3</td>
</tr>
<tr>
<td>-=</td>
<td>i -= 3</td>
<td>i = i - 3</td>
</tr>
<tr>
<td>*=</td>
<td>i *= 3</td>
<td>i = i * 3</td>
</tr>
<tr>
<td>/=</td>
<td>i /= 3</td>
<td>i = i / 3</td>
</tr>
<tr>
<td>^=</td>
<td>i ^= 3</td>
<td>i = i ^ 3</td>
</tr>
</tbody>
</table>

Unless you have prohibited the use of dashes in variable names, you must leave a space before -- and -=. If you do not, PowerScript reads the minus sign as part of a variable name. For more information, see “Identifier names” on page 5.

Examples

Example 1  These statements each assign a value to the variable ld_date:

```power
date ld_date
ld_date = Today( )
ld_date = 2006-01-01
ld_date = Date("January 1, 2006")
```

Example 2  These statements assign the parent of the current control to a window variable:

```power
window lw_current_window
lw_current_window = Parent
```

Example 3  This statement makes a CheckBox invisible:

```power
ck_on.Visible = FALSE
```

Example 4  This statement is not an assignment—it tests the value of the string in the SingleLineEdit sle_emp:

```power
IF sle_emp.Text = "N" THEN Open(win_1)
```
Assignment

Example 5  These statements concatenate two strings and assign the value to the string Text1:

    string Text1
    Text1 = sle_emp.Text + ".DAT"

Example 6  These assignments use operator shortcuts:

    int i = 4
    i ++    // i is now 5.
    i --    // i is 4 again.
    i += 10 // i is now 14.
    i /= 2  // i is now 7.

These shortcuts can be used only in pure assignment statements. They cannot be used with other operators in a statement. For example, the following is invalid:

    int i, j
    i = 12
    j = i ++    // INVALID

The following is valid, because ++ is used by itself in the assignment:

    int i, j
    i = 12
    i ++
    j = i
CALL

Description

Calls an ancestor script from a script for a descendent object. You can call scripts for events in an ancestor of the user object, menu, or window. You can also call scripts for events for controls in an ancestor of the user object or window.

When you use the CALL statement to call an ancestor event script, the AncestorReturnValue variable is generated. For more information on the AncestorReturnValue variable, see “About events” on page 183.

Syntax

CALL ancestorobject {'controlname'}::event

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestorobject</td>
<td>An ancestor of the descendent object</td>
</tr>
<tr>
<td>controlname</td>
<td>The name of a control in an ancestor window or custom user object (optional)</td>
</tr>
<tr>
<td>event</td>
<td>An event in the ancestor object</td>
</tr>
</tbody>
</table>

Usage

Using the standard syntax

For most purposes, you should use the standard syntax for calling functions and events. For more information about the standard syntax, see “Syntax for calling PowerBuilder functions and events” on page 111.

The standard syntax allows you to trigger or post an event or function in an ancestor and then pass arguments, but it does not allow you to call a script for a control in the ancestor.

In some circumstances, you can use the pronoun Super when ancestorobject is the descendant object’s immediate ancestor. See the discussion of “Super pronoun” on page 14.

If the call is being made to an ancestor event, the arguments passed to the current event are automatically propagated to the ancestor event. If you call a non-ancestor event and pass arguments, you need to use the new syntax, otherwise null will be passed for each argument.

Examples

Example 1 This statement calls a script for an event in an ancestor window:

CALL w_emp::Open

Example 2 This statement calls a script for an event in a control in an ancestor window:

CALL w_emp`cb_close::Clicked
**CHOOSE CASE**

**Description**
A control structure that directs program execution based on the value of a test expression (usually a variable).

**Syntax**
```
CHOOSE CASE testexpression
case expressionlist
statementblock
{ case expressionlist
statementblock
.
.
.
case expressionlist
statementblock
}
case ELSE
statementblock
end choose
```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>testexpression</td>
<td>The expression on which you want to base the execution of the script</td>
</tr>
<tr>
<td>expressionlist</td>
<td>One of the following expressions:</td>
</tr>
<tr>
<td></td>
<td>• A single value</td>
</tr>
<tr>
<td></td>
<td>• A list of values separated by commas (such as 2, 4, 6, 8)</td>
</tr>
<tr>
<td></td>
<td>• A TO clause (such as 1 TO 30)</td>
</tr>
<tr>
<td></td>
<td>• IS followed by a relational operator and comparison value (such as IS&gt;5)</td>
</tr>
<tr>
<td></td>
<td>• Any combination of the above with an implied OR between expressions (such as 1, 3, 5, 7, 9, 27 TO 33, IS &gt;42)</td>
</tr>
<tr>
<td>statementblock</td>
<td>The block of statements you want PowerBuilder to execute if the test expression matches the value in expressionlist</td>
</tr>
</tbody>
</table>

**Usage**
At least one CASE clause is required. You must end a CHOOSE CASE control structure with END CHOOSE.

If `testexpression` at the beginning of the CHOOSE CASE statement matches a value in `expressionlist` for a CASE clause, the statements immediately following the CASE clause are executed. Control then passes to the first statement after the END CHOOSE clause.

If multiple CASE expressions exist, then `testexpression` is compared to each `expressionlist` until a match is found or the CASE ELSE or END CHOOSE is encountered.
If there is a CASE ELSE clause and the test value does not match any of the expressions, *statementblock* in the CASE ELSE clause is executed. If no CASE ELSE clause exists and a match is not found, the first statement after the END CHOOSE clause is executed.

**Examples**  

**Example 1**  These statements provide different processing based on the value of the variable *Weight*:

```powerscript
CHOOSE
CASE Weight
CASE IS<16
  Postage=Weight*0.30
  Method="USPS"
CASE 16 to 48
  Postage=4.50
  Method="UPS"
CASE ELSE
  Postage=25.00
  Method="FedEx"
END CHOOSE
```

**Example 2**  These statements convert the text in a SingleLineEdit control to a real value and provide different processing based on its value:

```powerscript
CHOOSE
CASE Real(sle_real.Text)
CASE is < 10.99999
  sle_message.Text = "Real Case < 10.99999"
CASE 11.00 to 48.99999
  sle_message.Text = "Real Case 11 to 48.99999"
CASE is > 48.99999
  sle_message.Text = "Real Case > 48.99999"
CASE ELSE
  sle_message.Text = "Cannot evaluate!"
END CHOOSE
```
**CONTINUE**

**Description**
In a DO...LOOP or a FOR...NEXT control structure, skips statements in the loop. CONTINUE takes no parameters.

**Syntax**
CONTINUE

**Usage**
When PowerBuilder encounters a CONTINUE statement in a DO...LOOP or FOR...NEXT block, control passes to the next LOOP or NEXT statement. The statements between the CONTINUE statement and the loop’s end statement are skipped in the current iteration of the loop. In a nested loop, a CONTINUE statement bypasses statements in the current loop structure.

For information on how to break out of the loop, see EXIT on page 133.

**Examples**

**Example 1**
These statements display a message box twice: when \( B \) equals 2 and when \( B \) equals 3. As soon as \( B \) is greater than 3, the statement following CONTINUE is skipped during each iteration of the loop:

```powershell
integer A=1, B=1
DO WHILE A < 100
    A = A+1
    B = B+1
    IF B > 3 THEN CONTINUE
    MessageBox("Hi", "B is " + String(B) )
LOOP
```

**Example 2**
These statements stop incrementing \( B \) as soon as \( Count \) is greater than 15:

```powershell
integer A=0, B=0, Count
FOR Count = 1 to 100
    A = A + 1
    IF Count > 15 THEN CONTINUE
    B = B + 1
NEXT
// Upon completion, a=100 and b=15.
```
CREATE

Description

Creates an object instance for a specified object type. After a CREATE statement, properties of the created object instance can be referenced using dot notation.

The CREATE statement returns an object instance that can be stored in a variable of the same type.

Syntax 1 specifies the object type at compilation. Syntax 2 allows the application to choose the object type dynamically.

Syntax

Syntax 1 (specifies the object type at compilation):

\[
\text{objectvariable} = \text{CREATE} \ \text{objecttype}
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectvariable</td>
<td>A global, instance, or local variable whose datatype is objecttype</td>
</tr>
<tr>
<td>objecttype</td>
<td>The object datatype</td>
</tr>
</tbody>
</table>

Syntax 2 (allows the application to choose the object type dynamically):

\[
\text{objectvariable} = \text{CREATE USING} \ \text{objecttypestring}
\]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectvariable</td>
<td>A global, instance, or local variable whose datatype is the same class as the object being created or an ancestor of that class</td>
</tr>
<tr>
<td>objecttypestring</td>
<td>A string whose value is the name of the class datatype to be created</td>
</tr>
</tbody>
</table>

Usage

Use CREATE as the first reference to any class user object. This includes standard class user objects such as mailSession or Transaction.

The system provides one instance of several standard class user objects: Message, Error, Transaction, DynamicDescriptionArea, and DynamicStagingArea. You only need to use CREATE if you declare additional instances of these objects.

If you need a menu that is not part of an open window definition, use CREATE to create an instance of the menu. (See the function PopMenu on page 828.)

To create an instance of a visual user object or window, use the appropriate Open function (instead of CREATE).
You do not need to use CREATE to allocate memory for:

- A standard datatype, such as integer or string
- Any structure, such as the Environment object
- Any object whose AutoInstantiate setting is true
- Any object that has been instantiated using a function, such as Open

**Specifying the object type dynamically**  
CREATE USING allows your application to choose the object type dynamically. It is usually used to instantiate an ancestor variable with an instance of one of its descendants. The particular descendant is chosen at execution time.

For example, if `uo_a` has two descendants, `uo_a_desc1` and `uo_a_desc2`, then the application can select the object to be created based on current conditions:

```pascal
uo_a uo_a_var
string ls_objectname

IF ... THEN
    ls_objectname = "uo_a_desc1"
ELSE
    ls_objectname = "uo_a_desc2"
END IF
uo_a_var = CREATE USING ls_objectname
```

**Destroying objects you create**  
When you have finished with an object you created, you can call DESTROY to release its memory. However, you should call DESTROY only if you are sure that the object is not referenced by any other object. PowerBuilder’s garbage collection mechanism maintains a count of references to each object and destroys unreferenced objects automatically.

For more information about garbage collection, see “Garbage collection” on page 84.

**Examples**

**Example 1** These statements create a new transaction object and stores the object in the variable `DBTrans`:

```pascal
transaction DBTrans
DBTrans = CREATE transaction
DBTrans.DBMS = 'ODBC'
```

**Example 2** These statements create a user object when the application has need of the services it provides. Because the user object might or might not exist, the code that accesses it checks whether it exists before calling its functions.
The object that creates the service object declares `invo_service` as an instance variable:

```
  n_service invo_service
```

The Open event for the object creates the service object:

```
//Open event of some object
IF (some condition) THEN
   invo_service = CREATE n_service
END IF
```

When another script wants to call a function that belongs to the `n_service` class, it verifies that `invo_service` is instantiated:

```
IF IsValid(invo_service) THEN
   invo_service.of_perform_some_work()
END IF
```

If the service object was created, then it also needs to be destroyed:

```
IF isvalid(invo_service) THEN DESTROY invo_service
```

Example 3  When you create a DataStore object, you also have to give it a DataObject and call `SetTransObject` before you can use it:

```
l_ds_delete = CREATE u_ds
l_ds_delete.DataObject = 'd_user_delete'
l_ds_delete.SetTransObject(SQLCA)
l1_cnt = l_ds_delete.Retrieve(lstr_data.name)
```

Example 4  In this example, `n_file_service_class` is an ancestor object, and `n_file_service_class_ansi` and `n_file_service_class_dbcs` are its descendants. They hold functions and variables that provide services for the application. The code chooses which object to create based on whether the user is running in a DBCS environment:

```
n_file_service_class  lnv_fileservice
string ls_objectname
environment luo_env

GetEnvironment ( luo_env )
IF luo_env.charset = charsetdbcs! THEN
   ls_objectname = "n_file_service_class_dbcs"
ELSE
   ls_objectname = "n_file_service_class_ansi"
END IF

lnv_fileservice = CREATE USING ls_objectname
DESTROY

Description
Eliminates an object instance that was created with the CREATE statement. After a DESTROY statement, properties of the deleted object instance can no longer be referenced.

Syntax
DESTROY objectvariable

Parameter | Description
---|---
objectvariable | A variable whose datatype is a PowerBuilder object

Usage
When you are finished with an object that you created, you can call DESTROY to release its memory. However, you should call DESTROY only if you are sure that the object is not referenced by any other object. PowerBuilder’s garbage collection mechanism maintains a count of references to each object and destroys unreferenced objects automatically.

For more information about garbage collection, see “Garbage collection” on page 84.

All objects are destroyed automatically when your application terminates.

Examples
**Example 1** The following statement destroys the transaction object DBTrans that was created with a CREATE statement:

```
DESTROY DBTrans
```

**Example 2** This example creates an OLEStorage variable istg_prod_pic in a window’s Open event. When the window is closed, the Close event script destroys the object. The variable’s declaration is:

```
OLEStorage istg_prod_pic
```

The window’s Open event creates an object instance and opens an OLE storage file:

```
integer li_result
istg_prod_pic = CREATE OLEStorage
li_result = stg_prod_pic.Open("PICTURES.OLE")
```

The window’s Close event destroys *istg_prod_pic*:

```
integer li_result
li_result = istg_prod_pic.Save( )
IF li_result = 0 THEN
    DESTROY istg_prod_pic
END IF
```
DO...LOOP

Description

A control structure that is a general-purpose iteration statement used to execute a block of statements while or until a condition is true.

DO...LOOP has four formats:

• **DO UNTIL**  Executes a block of statements until the specified condition is true. If the condition is true on the first evaluation, the statement block does not execute.

• **DO WHILE**  Executes a block of statements while the specified condition is true. The loop ends when the condition becomes false. If the condition is false on the first evaluation, the statement block does not execute.

• **LOOP UNTIL**  Executes a block of statements at least once and continues until the specified condition is true.

• **LOOP WHILE**  Executes a block of statements at least once and continues while the specified condition is true. The loop ends when the condition becomes false.

In all four formats of the DO...LOOP control structure, **DO** marks the beginning of the statement block that you want to repeat. The **LOOP** statement marks the end.

You can nest DO...LOOP control structures.

Syntax

```
DO UNTIL condition
  statementblock
LOOP
DO WHILE condition
  statementblock
LOOP
DO
  statementblock
LOOP UNTIL condition
DO
  statementblock
LOOP WHILE condition
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>The condition you are testing</td>
</tr>
<tr>
<td>statementblock</td>
<td>The block of statements you want to repeat</td>
</tr>
</tbody>
</table>
DO...LOOP

Usage

Use DO WHILE or DO UNTIL when you want to execute a block of statements only if a condition is true (for WHILE) or false (for UNTIL). DO WHILE and DO UNTIL test the condition before executing the block of statements.

Use LOOP WHILE or LOOP UNTIL when you want to execute a block of statements at least once. LOOP WHILE and LOOP UNTIL test the condition after the block of statements has been executed.

Examples

DO UNTIL  The following DO UNTIL repeatedly executes the Beep function until A is greater than 15:

```PowerBuilder
integer A = 1, B = 1
DO UNTIL A > 15
   Beep(A)
   A = (A + 1) * B
LOOP
```

DO WHILE  The following DO WHILE repeatedly executes the Beep function only while A is less than or equal to 15:

```PowerBuilder
integer A = 1, B = 1
DO WHILE A <= 15
   Beep(A)
   A = (A + 1) * B
LOOP
```

LOOP UNTIL  The following LOOP UNTIL executes the Beep function and then continues to execute the function until A is greater than 1:

```PowerBuilder
integer A = 1, B = 1
DO
   Beep(A)
   A = (A + 1) * B
LOOP UNTIL A > 15
```

LOOP WHILE  The following LOOP WHILE repeatedly executes the Beep function while A is less than or equal to 15:

```PowerBuilder
integer A = 1, B = 1
DO
   Beep(A)
   A = (A + 1) * B
LOOP WHILE A <= 15
```
EXIT

Description
In a DO...LOOP or a FOR...NEXT control structure, passes control out of the current loop. EXIT takes no parameters.

Syntax
EXIT

Usage
An EXIT statement in a DO...LOOP or FOR...NEXT control structure causes control to pass to the statement following the LOOP or NEXT statement. In a nested loop, an EXIT statement passes control out of the current loop structure.

For information on how to jump to the end of the loop and continue looping, see CONTINUE on page 126.

Examples

**Example 1**  This EXIT statement causes the loop to terminate if an element in the Nbr array equals 0:

```
int Nbr[10]
int Count = 1
// Assume values get assigned to Nbr array...

DO WHILE Count < 11
    IF Nbr[Count] = 0 THEN EXIT
    Count = Count + 1
LOOP

MessageBox("Hi", "Count is now " + String(Count) )
```

**Example 2**  This EXIT statement causes the loop to terminate if an element in the Nbr array equals 0:

```
int Nbr[10]
int Count
// Assume values get assigned to Nbr array...

FOR Count = 1 to 10
    IF Nbr[Count] = 0 THEN EXIT
NEXT

MessageBox("Hi", "Count is now " + String(Count) )
```
FOR...NEXT

Description
A control structure that is a numerical iteration, used to execute one or more statements a specified number of times.

Syntax
FOR varname = start TO end (STEP increment)
  statementblock
NEXT

Parameter | Description
---|---
varname | The name of the iteration counter variable. It can be any numerical type (byte, integer, double, real, long, longlong, or decimal), but integers provide the fastest performance.
start | Starting value of varname.
end | Ending value of varname.
increment (optional) | The increment value. Increment must be a constant and the same datatype as varname. If you enter an increment, STEP is required. +1 is the default increment.
statementblock | The block of statements you want to repeat.

Usage

Using the start and end parameters  For a positive increment, end must be greater than start. For a negative increment, end must be less than start.

When increment is positive and start is greater than end, statementblock does not execute. When increment is negative and start is less than end, statementblock does not execute.

When start and end are expressions, they are reevaluated on each pass through the loop. If the expression’s value changes, it affects the number of loops.
Consider this example—the body of the loop changes the number of rows, which changes the result of the RowCount function:

```
FOR n = 1 TO dw_1.RowCount()
  dw_1.DeleteRow(1)
NEXT
```

A variable as the step increment
If you need to use a variable for the step increment, you can use one of the DO...LOOP constructions and increment the counter yourself within the loop.

Nesting  You can nest FOR...NEXT statements. You must have a NEXT for each FOR.
You can end the FOR loop with the keywords END FOR instead of NEXT.
Avoid overflow
If start or end is too large for the datatype of varname, varname will overflow, which might create an infinite loop. Consider this statement for the integer li_int:

```
FOR li_int = 1 TO 50000
```

The end value 50000 is too large for an integer. When li_int is incremented, it overflows to a negative value before reaching 50000, creating an infinite loop.

Examples

**Example 1** These statements add 10 to A as long as n is >=5 and <=25:

```
FOR n = 5 to 25
  A = A+10
NEXT
```

**Example 2** These statements add 10 to A and increment n by 5 as long as n is >=5 and <=25:

```
FOR N = 5 TO 25 STEP 5
  A = A+10
NEXT
```

**Example 3** These statements contain two lines that will never execute because increment is negative and start is less than end:

```
FOR Count = 1 TO 100 STEP -1
  IF Count < 1 THEN EXIT // These 2 lines
  Box[Count] = 10        // will never execute.
NEXT
```

**Example 4** These are nested FOR...NEXT statements:

```
Int Matrix[100,50,200]
FOR i = 1 to 100
  FOR j = 1 to 50
    FOR k = 1 to 200
      Matrix[i,j,k]=1
    NEXT
  NEXT
NEXT
```
**GOTO**

**Description**
Transfers control from one statement in a script to another statement that is labeled.

**Syntax**
GOTO label

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>The label associated with the statement to which you want to transfer control. A label is an identifier followed by a colon (such as OK:). Do not use the colon with a label in the GOTO statement.</td>
</tr>
</tbody>
</table>

**Examples**

**Example 1**  This GOTO statement skips over the Taxable=FALSE line:

```
Goto NextStep
Taxable=FALSE  //This statement never executes.
NextStep:
Rate=Count/Count4
```

**Example 2**  This GOTO statement transfers control to the statement associated with the label OK:

```
GOTO OK
```

HALT

Description
Terminates an application.

Syntax
HALT {CLOSE}

Usage
When PowerBuilder encounters Halt without the keyword CLOSE, it immediately terminates the application.

When PowerBuilder encounters Halt with the keyword CLOSE, it immediately executes the scripts for application Close event and for the CloseQuery, Close, and Destructor events on all instantiated objects before terminating the application. If there are no scripts for these events, PowerBuilder immediately terminates the application.

You should not code a HALT statement in a component that will run in a server environment. When a PowerBuilder component is running in a server such as EAServer or J2EE, and a HALT statement is encountered, instead of aborting the application, which is in this case the server itself, the PowerBuilder VM throws a runtime error and continues. The container is responsible for managing the lifecycle of the component. In EAServer, the error message is written to the Jaguar log, even if the runtime error causes a transaction rollback and the transaction is overridden by a new transaction.

Examples

Example 1  This statement stops the application if the user enters a password in the SingleLineEdit named sle_password that does not match the value stored in a string named CorrectPassword:

```powerbuilder
IF sle_password.Text <> CorrectPassword THEN HALT
```

Example 2  This statement executes the script for the Close event for the application before it terminates the application if the user enters a password in sle_password that does not match the value stored in the string CorrectPassword:

```powerbuilder
IF sle_password.Text <> CorrectPassword &
    THEN HALT CLOSE
```
**IF...THEN**

**Description**
A control structure used to cause a script to perform a specified action if a stated condition is true. Syntax 1 uses a single-line format, and Syntax 2 uses a multiline format.

**Syntax**

Syntax 1 (the single-line format):

```plaintext
IF condition THEN action1 {ELSE action2}
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>condition</code></td>
<td>The condition you want to test.</td>
</tr>
<tr>
<td><code>action1</code></td>
<td>The action you want performed if the condition is true. The action must be a single statement on the same line as the rest of the IF statement.</td>
</tr>
<tr>
<td><code>action2</code></td>
<td>(optional) The action you want performed if the condition is false. The action must be a single statement on the same line as the rest of the IF statement.</td>
</tr>
</tbody>
</table>

Syntax 2 (the multiline format):

```plaintext
IF condition1 THEN action1
ELSEIF condition2 THEN action2
.
.
ELSE action3
END IF
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>condition1</code></td>
<td>The first condition you want to test.</td>
</tr>
<tr>
<td><code>action1</code></td>
<td>The action you want performed if <code>condition1</code> is true. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines. At least one action is required.</td>
</tr>
<tr>
<td><code>condition2</code></td>
<td>(optional) The condition you want to test if <code>condition1</code> is false. You can have multiple ELSEIF...THEN statements in an IF...THEN control structure.</td>
</tr>
<tr>
<td><code>action2</code></td>
<td>The action you want performed if <code>condition2</code> is true. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines.</td>
</tr>
<tr>
<td><code>action3</code></td>
<td>(optional) The action you want performed if none of the preceding conditions is true. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines.</td>
</tr>
</tbody>
</table>
Usage

You can use continuation characters to place the single-line format on more than one physical line in the script.

You must end a multiline IF...THEN control structure with END IF (which is two words).

Examples

**Example 1**  This single-line IF...THEN statement opens window w_first if Num is equal to 1; otherwise, w_rest is opened:

```
IF Num = 1 THEN Open(w_first) ELSE Open(w_rest)
```

**Example 2**  This single-line IF...THEN statement displays a message if the value in the SingleLineEdit sle_State is “TX”. It uses the continuation character to continue the single-line statement across two physical lines in the script:

```
IF sle_State.text="TX" THEN   &
   MessageBox("Hello","Tex")
```

**Example 3**  This multiline IF...THEN compares the horizontal positions of windows w_first and w_second. If w_first is to the right of w_second, w_first is moved to the left side of the screen:

```
IF w_first.X > w_second.X THEN
   w_first.X = 0
END IF
```

**Example 4**  This multiline IF...THEN causes the application to:

- Beep twice if X equals Y
- Display the Parts list box and highlight item 5 if X equals Z
- Display the Choose list box if X is blank
- Hide the Empty button and display the Full button if none of the above conditions is true

```
IF X=Y THEN
   Beep(2)
ELSEIF X=Z THEN
   Show (lb_parts); lb_parts.SetState(5,TRUE)
ELSEIF X=" " THEN
   Show (lb_choose)
ELSE
   Hide(cb_empty)
   Show(cb_full)
END IF
```
RETURN

Description
Stops the execution of a script or function immediately.

Syntax
RETURN { expression }

Parameter | Description
--- | ---
expression | In a function, any value (or expression) you want the function to return. The return value must be the datatype specified as the return type in the function.

Usage
When a user’s action triggers an event and PowerBuilder encounters RETURN in the event script, it terminates execution of that script immediately and waits for the next user action.

When a script calls a function or event and PowerBuilder encounters RETURN in the code, RETURN transfers (returns) control to the point at which the function or event was called.

Examples
Example 1 This script causes the system to beep once; the second beep statement will not execute:

```plaintext
Beep(1)
RETURN
Beep(1)  // This statement will not execute.
```

Example 2 These statements in a user-defined function return the result of dividing Arg1 by Arg2 if Arg2 is not equal to zero; they return -1 if Arg2 is equal to zero:

```plaintext
IF Arg2 <> 0 THEN
   RETURN Arg1/Arg2
ELSE
   RETURN -1
END IF
```
THROW

Description

Used to manually trigger exception handling for user-defined exceptions.

Syntax

THROW exlvalue

Parameter | Description
--- | ---
exlvalue | Variable (or expression that evaluates to a valid instance of an object) of type Throwable. Usually the object type thrown is a user-defined exception class derived from the system Exception class that inherits from Throwable.

Usage

The variable following the THROW reserved word must be a valid object instance or an expression that produces a valid object instance that derives from the Throwable datatype. For example, you can use an expression such as:

```
THROW create ExceptionType
```

where ExceptionType is an object of type Throwable.

If you attempt to throw a noninstantiated exception, you will not get back the exception information you want, since the only exception information you retrieve will be a NullObjectError.

In a method script, you can only throw an exception that you declare in the method prototype or that you handle in a try-catch block. The PowerScript compiler displays an error message if you try to throw a user-defined exception without declaring it in the prototype Throws statement and without surrounding it in an appropriate try-catch block.

When a RuntimeException, or a descendant of RuntimeException, is thrown, the instance variable containing line number information will be filled in at the point where the THROW statement occurs. If the error is handled and thrown again, this information will not be updated unless it has specifically been set to null.

Examples

```
long ll_result
ll_result = myConnection.ConnectToServer()

ConnectionException ex
ex = create ConnectionException
ex.connectResult = ll_result
THROW ex
end if
```
THROWS

Description
Used to declare the type of exception that a method triggers. It is part of the method prototype.

Syntax
```
methodname ( { arguments } ) THROWS ExceptionType { , ExceptionType, ... }
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>methodname</td>
<td>Name of the method that throws an exception.</td>
</tr>
<tr>
<td>arguments</td>
<td>Arguments of the method that throws an exception. Depending on the method, the method arguments can be optional.</td>
</tr>
<tr>
<td>ExceptionType</td>
<td>Object of type Throwable. Usually the object type thrown is a user-defined exception class derived from the system Exception class. If you define multiple potential exceptions for a method, you can throw each type of exception in the same clause by separating the exception types with commas.</td>
</tr>
</tbody>
</table>

Usage
Internal use only.

You do not type or otherwise add the THROWS clause to function calls in a PowerBuilder script. However, you can add a THROWS clause to any PowerBuilder function or to any user event that is not defined by a pbm event ID.

For more information about adding a THROWS clause to a function or event prototype, see the PowerBuilder Users Guide. For more information about exception handling, see Application Techniques.
TRY...CATCH...FINALLY...END TRY

Description
Isolates code that can cause an exception, describes what to do if an exception of a given type is encountered, and allows you to close files or network connections (and return objects to their original state) whether or not an exception is encountered.

Syntax
TRY
  trystatements
  CATCH ( ThrowableType1 exIdentifier1 )
    catchstatements1
  CATCH ( ThrowableType2 exIdentifier2 )
    catchstatements2
  ...
  CATCH ( ThrowableTypeN exIdentifierN )
    catchstatementsN
FINALLY
  cleanupstatements
END TRY

Parameter | Description
--- | ---
trystatements | Block of code that might potentially throw an exception.
ThrowableTypeN | Object type of exception to be caught. A CATCH block is optional if you include a FINALLY block. You can include multiple CATCH blocks. Every CATCH block in a try-catch block must include a corresponding exception object type and a local variable of that type.
exIdentifierN | Local variable of type ThrowableTypeN.
catchstatementsN | Code to handle the exception being caught.
cleanupstatements | Cleanup code. The FINALLY block is optional if you include one or more CATCH block.

Usage
The TRY block, which is the block of statements between the TRY and CATCH keywords (or the TRY and FINALLY keywords if there is no CATCH clause), is used to isolate code that might potentially throw an exception. The statements in the TRY block are run unconditionally until either the entire block of statements is executed or some statement in the block causes an exception to be thrown.

Use a CATCH block or multiple CATCH blocks to handle exceptions thrown in a TRY block. In the event that an exception is thrown, execution of the TRY block is stopped and the statements in the first CATCH block are executed—if and only if the exception thrown is of the same type or a descendant of the type of the identifier following the CATCH keyword.
TRY...CATCH...FINALLY...END TRY

If the exception thrown is not the same type or a descendant type of the identifier in the first CATCH block, the exception is not handled by this CATCH block. If there are additional CATCH blocks, they are evaluated in the order they appear. If the exception cannot be handled by any of the CATCH blocks, the statements in the FINALLY block are executed.

The exception then continues to unwind the call stack to any outer nested try-catch blocks. If there are no outer nested blocks, the SystemError event on the Application object is fired.

If no exception is thrown, execution continues at the beginning of the FINALLY block if one exists; otherwise, execution continues on the line following the END TRY statement.

See also

THROW
CHAPTER 8 SQL Statements

About this chapter

This chapter describes the embedded SQL and dynamic SQL statements and how to use them in scripts.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using SQL in scripts</td>
<td>146</td>
</tr>
<tr>
<td>CLOSE Cursor</td>
<td>149</td>
</tr>
<tr>
<td>CLOSE Procedure</td>
<td>150</td>
</tr>
<tr>
<td>COMMIT</td>
<td>151</td>
</tr>
<tr>
<td>CONNECT</td>
<td>152</td>
</tr>
<tr>
<td>DECLARE Cursor</td>
<td>153</td>
</tr>
<tr>
<td>DECLARE Procedure</td>
<td>153</td>
</tr>
<tr>
<td>DELETE</td>
<td>155</td>
</tr>
<tr>
<td>DELETE Where Current of Cursor</td>
<td>156</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>156</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>157</td>
</tr>
<tr>
<td>FETCH</td>
<td>158</td>
</tr>
<tr>
<td>INSERT</td>
<td>159</td>
</tr>
<tr>
<td>OPEN Cursor</td>
<td>160</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>160</td>
</tr>
<tr>
<td>SELECT</td>
<td>161</td>
</tr>
<tr>
<td>SELECTBLOB</td>
<td>162</td>
</tr>
<tr>
<td>UPDATE</td>
<td>164</td>
</tr>
<tr>
<td>UPDATEBLOB</td>
<td>165</td>
</tr>
<tr>
<td>UPDATE Where Current of Cursor</td>
<td>166</td>
</tr>
<tr>
<td>Using dynamic SQL</td>
<td>167</td>
</tr>
<tr>
<td>Dynamic SQL Format 1</td>
<td>171</td>
</tr>
<tr>
<td>Dynamic SQL Format 2</td>
<td>172</td>
</tr>
<tr>
<td>Dynamic SQL Format 3</td>
<td>173</td>
</tr>
<tr>
<td>Dynamic SQL Format 4</td>
<td>176</td>
</tr>
</tbody>
</table>
Using SQL in scripts

PowerScript supports standard embedded SQL statements and dynamic SQL statements in scripts. In general, PowerScript supports all DBMS-specific clauses and reserved words that occur in the supported SQL statements. For example, PowerBuilder supports DBMS-specific built-in functions within a SELECT command.

For information about embedded SQL, see online Help.

Wherever constants can be referenced in SQL statements, PowerScript variables preceded by a colon (:) can be substituted. Any valid PowerScript variable can be used. This INSERT statement uses a constant value:

```
INSERT INTO EMPLOYEE ( SALARY )
VALUES ( 18900 ) ;
```

The same statement using a PowerScript variable to reference the constant might look like this:

```
int   Sal_var
Sal_var = 18900
INSERT INTO EMPLOYEE ( SALARY )
VALUES ( :Sal_var ) ;
```

PowerBuilder supports indicator variables, which are used to identify null values or conversion errors after a database retrieval. Indicator variables are integers that are specified in the HostVariableList of a FETCH or SELECT statement.

Each indicator variable is separated from the variable it is indicating by a space (but no comma). For example, this statement is a HostVariableList without indicator variables:

```
:Name, :Address, :City
```

The same HostVariableList with indicator variables looks like this:

```
:Name :IndVar1, :Address :IndVar2, :City :IndVar3
```

Indicator variables have one of these values:

<table>
<thead>
<tr>
<th>Page</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Valid, non-null value</td>
</tr>
<tr>
<td>-1</td>
<td>Null value</td>
</tr>
<tr>
<td>-2</td>
<td>Conversion error</td>
</tr>
</tbody>
</table>
Error reporting
Not all DBMSs return a conversion error when the datatype of a column does not match the datatype of the associated variable.

The following statement uses the indicator variable `IndVar2` to see if `Address` contains a null value:

```
if IndVar2 = -1 then...
```

You can also use the PowerScript `IsNull` function to accomplish the same result without using indicator variables:

```
if IsNull(Address) then ...
```

This statement uses the indicator variable `IndVar3` to set `City` to null:

```
IndVar3 = -1
```

You can also use the PowerScript `SetNull` function to accomplish the same result without using indicator variables:

```
SetNull(City)
```

Error handling in scripts
The scripts shown in the SQL examples above do not include error handling, but it is good practice to test the success and failure codes (the `SQLCode` attribute) in the transaction object after every statement. The codes are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success.</td>
</tr>
<tr>
<td>100</td>
<td>Fetched row not found.</td>
</tr>
<tr>
<td>-1</td>
<td>Error; the statement failed. Use <code>SQLErrText</code> or <code>SQLDBCode</code> to obtain the detail.</td>
</tr>
</tbody>
</table>

After certain statements, such as DELETE, FETCH, and UPDATE, you should also check the `SQLNRRows` property of the transaction object to make sure the action affected at least one row.

About `SQLErrText` and `SQLDBCode` The string `SQLErrText` in the transaction object contains the database vendor-supplied error message. The long named `SQLDBCode` in the transaction object contains the database vendor-supplied status code:

```pascal
IF SQLCA.SQLCode = -1 THEN
    MessageBox("SQL error", SQLCA.SQLErrText)
END IF
```
You can paint the following SQL statements in scripts and functions:

- Declarations of SQL cursors and stored procedures
- Cursor FETCH, UPDATE, and DELETE statements
- Noncursor SELECT, INSERT, UPDATE, and DELETE statements

For more information about scope, see “Where to declare variables” on page 34.

You can declare cursors and stored procedures at the scope of global, instance, shared, or local variables. A cursor or procedure can be declared in the Script view using the Paste SQL button in the PainterBar.

You can paint standard embedded SQL statements in the Script view, the Function painter, and the Interactive SQL view in the Database painter using the Paste SQL button in the PainterBar or the Paste Special>SQL item from the pop-up menu.

In general, all DBMS-specific features are supported in PowerScript if they occur within a PowerScript-supported SQL statement. For example, PowerScript supports DBMS-specific built-in functions within a SELECT command.

However, any SQL statement that contains a SELECT clause must also contain a FROM clause in order for the script to compile successfully. To solve this problem, add a FROM clause that uses a “dummy” table to SELECT statements without FROM clauses. For example:

```sql
string res
select user_name() into:res from dummy;
select db_name() into:res from dummy;
selct date('2001-01-02:21:20:53') into:res from dummy;
```

When PowerBuilder compiles an application that contains embedded SQL, it connects to the database profile last used in order to check for database access errors during the build process. For applications that use multiple databases, this can result in spurious warnings during the build since the embedded SQL can be validated only against that single last-used database and not against the databases actually used by the application. In addition, an unattended build, such as a lengthy overnight rebuild, can stall if the database connection cannot be made.

To avoid these issues, you can select the Disable Database Connection When Compiling and Building check box on the general page of the System Options dialog box.
Caution
Select the check box only when you want to compile without signing on to the database. Compiling without connecting to a database prevents the build process from checking for database errors and may therefore result in runtime errors later.

CLOSE Cursor

Description
Closes the SQL cursor *CursorName*; ends processing of *CursorName*.

Syntax
```
CLOSE *CursorName* ;
```

Parameter | Description
--- | ---
*CursorName* | The cursor you want to close

Usage
This statement must be preceded by an OPEN statement for the same cursor. The USING TransactionObject clause is not allowed with CLOSE; the transaction object was specified in the statement that declared the cursor.

CLOSE often appears in the script that is executed when the SQL code after a fetch equals 100 (not found).

Error handling
It is good practice to test the success/failure code after executing a CLOSE cursor statement.

Examples
This statement closes the *Emp_cursor* cursor:

```
CLOSE *Emp_cursor* ;
```
CLOSE Procedure

Description
Closes the SQL procedure *ProcedureName*; ends processing of *ProcedureName*.

**DBMS-specific**
Not all DBMSs support stored procedures.

Syntax
CLOSE *ProcedureName*;

Usage
This statement must be preceded by an EXECUTE statement for the same procedure. The USING TransactionObject clause is not allowed with CLOSE; the transaction object was specified in the statement that declared the procedure.

Use CLOSE only to close procedures that return result sets. PowerBuilder automatically closes procedures that do not return result sets (and sets the return code to 100).

CLOSE often appears in the script that is executed when the SQL code after a fetch equals 100 (not found).

**Error handling**
It is good practice to test the success/failure code after executing a CLOSE Procedure statement.

Examples
This statement closes the stored procedure named *Emp_proc*:

```
CLOSE Emp_proc ;
```
**COMMIT**

**Description**
Permanently updates all database operations since the previous COMMIT, ROLLBACK, or CONNECT for the specified transaction object.

**Using COMMIT and ROLLBACK in a server component**
COMMIT and ROLLBACK commands embedded in a server component might have different effects depending on the setting of the UseContextObject DBParm parameter.

For information on the UseContextObject parameter see *Connecting to Your Database*. For information on deploying components to a transaction server, see *Application Techniques*.

**Syntax**

```
COMMIT {USING TransactionObject};
```

**Parameter** | **Description**
--- | ---
*TransactionObject* | The name of the transaction object for which you want to permanently update all database operations since the previous COMMIT, ROLLBACK, or CONNECT. This clause is required only for transaction objects other than the default (SQLCA).

**Usage**
COMMIT does not cause a disconnect, but it does close all open cursors or procedures. (But note that the DISCONNECT statement in PowerBuilder does issue a COMMIT.)

**Error handling**
It is good practice to test the success/failure code after executing a COMMIT statement.

**Examples**

**Example 1**  
This statement commits all operations for the database specified in the default transaction object:

```powershell
COMMIT ;
```

**Example 2**  
This statement commits all operations for the database specified in the transaction object named *Emp_tran*:

```powershell
COMMIT USING Emp_tran ;
```
CONNECT

Description
Connects to a specified database.

Syntax
CONNECT (USING TransactionObject);

Usage
This statement must be executed before any actions (such as INSERT, UPDATE, or DELETE) can be processed using the default transaction object or the specified transaction object.

Error handling
It is good practice to test the success/failure code after executing a CONNECT statement.

Examples
Example 1  This statement connects to the database specified in the default transaction object:

    CONNECT ;

Example 2  This statement connects to the database specified in the transaction object named Emp_tran:

    CONNECT USING Emp_tran ;

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransactionObject</td>
<td>The name of the transaction object containing the required connection information for the database to which you want to connect. This clause is required only for transaction objects other than the default (SQLCA).</td>
</tr>
</tbody>
</table>
DECLARE Cursor

**Description**
Declares a cursor for the specified transaction object.

**Syntax**
```
DECLARE CursorName CURSOR FOR SelectStatement
                {USING TransactionObject};
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CursorName</td>
<td>Any valid PowerBuilder name.</td>
</tr>
<tr>
<td>SelectStatement</td>
<td>Any valid SELECT statement.</td>
</tr>
<tr>
<td>TransactionObject</td>
<td>The name of the transaction object for which you want to declare the cursor. This clause is required only for transaction objects other than the default (SQLCA).</td>
</tr>
</tbody>
</table>

**Usage**
DECLARE Cursor is a nonexecutable command and is analogous to declaring a variable.

To declare a local cursor, open the script in the Script view and select Paste SQL from the PainterBar or the Edit>Paste Special menu. To declare a global, instance, or shared cursor, select Declare from the first drop-down list in the Script view and Global Variables, Instance Variables, or Shared Variables from the second drop-down list, then select Paste SQL.

For information about global, instance, shared, and local scope, see “Where to declare variables” on page 34.

**Examples**
This statement declares the cursor called `Emp_cur` for the database specified in the default transaction object. It also references the `Sal_var` variable, which must be set to an appropriate value before you execute the OPEN Emp_cur command:

```
DECLARE Emp_cur CURSOR FOR
    SELECT employee.emp_number, employee.emp_name
    FROM employee
    WHERE employee.emp_salary > :Sal_var ;
```

DECLARE Procedure

**Description**
Declares a procedure for the specified transaction object.

**DBMS-specific**
Not all DBMSs support stored procedures.
DECLARE Procedure

Syntax

DECLARE ProcedureName PROCEDURE FOR
   StoredProcedureName
   @Param1=Value1, @Param2=Value2,...
   {USING TransactionObject};

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcedureName</td>
<td>Any valid PowerBuilder name.</td>
</tr>
<tr>
<td>StoredProcedureName</td>
<td>Any stored procedure in the database.</td>
</tr>
<tr>
<td>@Paramn=Valuen</td>
<td>The name of a parameter (argument) defined in the stored procedure and a valid PowerBuilder expression; represents the number of the parameter and value.</td>
</tr>
<tr>
<td>TransactionObject</td>
<td>The name of the transaction object for which you want to declare the procedure. This clause is required only for transaction objects other than the default (SQLCA).</td>
</tr>
</tbody>
</table>

Usage

DECLARE Procedure is a nonexecutable command. It is analogous to declaring a variable.

To declare a local procedure, open the script in the Script view and select Paste SQL from the PainterBar or the Edit>Paste Special menu. To declare a global, instance, or shared procedure, select Declare from the first drop-down list in the Script view and Global Variables, Instance Variables, or Shared Variables from the second drop-down list, then select Paste SQL.

For information about global, instance, shared, and local scope, see “Where to declare variables” on page 34.

Examples

Example 1   This statement declares the Sybase ASE procedure Emp_proc for the database specified in the default transaction object. It references the Emp_name_var and Emp_sal_var variables, which must be set to appropriate values before you execute the EXECUTE Emp_proc command:

   DECLARE Emp_proc procedure for GetName
   @emp_name = :Emp_name_var,
   @emp_salary = :Emp_sal_var ;

Example 2   This statement declares the ORACLE procedure Emp_proc for the database specified in the default transaction object. It references the Emp_name_var and Emp_sal_var variables, which must be set to appropriate values before you execute the EXECUTE Emp_proc command:

   DECLARE Emp_proc procedure for GetName
   (:Emp_name_var, :Emp_sal_var) ;
**DELETE**

Description
Delete the rows in *TableName* specified by *Criteria*.

Syntax
```plaintext
DELETE FROM TableName WHERE Criteria {USING TransactionObject};
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TableName</code></td>
<td>The name of the table from which you want to delete rows.</td>
</tr>
<tr>
<td><code>Criteria</code></td>
<td>Criteria that specify which rows to delete.</td>
</tr>
<tr>
<td><code>TransactionObject</code></td>
<td>The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).</td>
</tr>
</tbody>
</table>

Usage

**Error handling**

It is good practice to test the success/failure code after executing a DELETE statement. To see if the DELETE was successful, you can test SQLCode for a failure code. However, if nothing matches the WHERE clause and no rows are deleted, SQLCode is still set to zero. To make sure the delete affected at least one row, check the SQLNRows property of the transaction object.

Examples

**Example 1**
This statement deletes rows from the Employee table in the database specified in the default transaction object where Emp_num is less than 100:

```plaintext
DELETE FROM Employee WHERE Emp_num < 100 ;
```

**Example 2**
These statements delete rows from the Employee table in the database named in the transaction object named `Emp_tran` where Emp_num is equal to the value entered in the `SingleLineEdit sle_number`:

```plaintext
int Emp_num
    Emp_num = Integer(sle_number.Text)
    DELETE FROM Employee
        WHERE Employee.Emp_num = :Emp_num ;
```

The integer *Emp_num* requires a colon in front of it to indicate it is a variable when it is used in a WHERE clause.
DELETE Where Current of Cursor

Description Deletes the row in which the cursor is positioned.

DBMS-specific Not all DBMSs support DELETE Where Current of Cursor.

Syntax DELETE FROM TableName WHERE CURRENT OF CursorName;

Parameter Description
---
TableName The name of the table from which you want to delete a row
CursorName The name of the cursor in which the table was specified

Usage The USING TransactionObject clause is not allowed with this form of DELETE Where Current of Cursor; the transaction object was specified in the statement that declared the cursor.

Error handling It is good practice to test the success/failure code after executing a DELETE Where Current of Cursor statement.

Examples This statement deletes from the Employee table the row in which the cursor named Emp_cur is positioned:

    DELETE FROM Employee WHERE current of Emp_curs ;

DISCONNECT

Description Executes a COMMIT for the specified transaction object and then disconnects from the specified database.

Syntax DISCONNECT {USING TransactionObject};

Parameter Description
---
TransactionObject The name of the transaction object that identifies the database you want to disconnect from and in which you want to permanently update all database operations since the previous COMMIT, ROLLBACK, or CONNECT. This clause is required only for transaction objects other than the default (SQLCA).
Usage

**Error handling**
It is good practice to test the success/failure code after executing a
DISCONNECT statement.

Examples

**Example 1** This statement disconnects from the database specified in the
default transaction object:

```
DISCONNECT ;
```

**Example 2** This statement disconnects from the database specified in the
transaction object named `Emp_tran`:

```
DISCONNECT USING Emp_tran ;
```

---

**EXECUTE**

**Description**
Executes the previously declared procedure identified by `ProcedureName`.

**Syntax**
EXECUTE `ProcedureName`;

**Parameter** | **Description**
---|---
`ProcedureName` | The name assigned in the DECLARE statement of the
stored procedure you want to execute. The procedure
must have been declared previously. `ProcedureName` is
not necessarily the name of the procedure stored in the
database.

**Usage**
The USING TransactionObject clause is not allowed with EXECUTE; the
transaction object was specified in the statement that declared the procedure.

**Error handling**
It is good practice to test the success/failure code after executing an EXECUTE statement.

**Examples**
This statement executes the stored procedure `Emp_proc`:

```
EXECUTE Emp_proc ;
```
FETCH

Description
Fetches the row after the row on which Cursor | Procedure is positioned.

Syntax
FETCH Cursor | Procedure INTO HostVariableList;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor or Procedure</td>
<td>The name of the cursor or procedure from which you want to fetch a row</td>
</tr>
<tr>
<td>HostVariableList</td>
<td>PowerScript variables into which data values will be retrieved</td>
</tr>
</tbody>
</table>

Usage
The USING TransactionObject clause is not allowed with FETCH; the transaction object was specified in the statement that declared the cursor or procedure.

If your DBMS supports formats of FETCH other than the customary (and default) FETCH NEXT, you can specify FETCH FIRST, FETCH PRIOR, or FETCH LAST.

Error handling
It is good practice to test the success/failure code after executing a FETCH statement. To see if the FETCH was successful, you can test SQLCode for a failure code. However, if nothing matches the WHERE clause and no rows are fetched, SQLCode is still set to 100. To make sure the fetch affected at least one row, check the SQLNRows property of the transaction object.

Examples
Example 1  This statement fetches data retrieved by the SELECT clause in the declaration of the cursor named Emp_cur and puts it into Emp_num and Emp_name:

```powerbuilder
int Emp_num
string Emp_name
FETCH Emp_cur INTO :Emp_num, :Emp_name;
```

Example 2  If sle_emp_num and sle_emp_name are SingleLineEdits, these statements fetch from the cursor named Emp_cur, store the data in Emp_num and Emp_name, and then convert Emp_num from an integer to a string, and put them in sle_emp_num and sle_emp_name:

```powerbuilder
int Emp_num
string Emp_name
FETCH Emp_cur INTO :emp_num, :emp_name;
sle_emp_num.Text = string(Emp_num)
sle_emp_name.Text = Emp_name
```
INSERT

Description
Inserts one or more new rows into the table specified in RestOfInsertStatement.

Syntax
```
INSERT RestOfInsertStatement
{USING TransactionObject} ;
```

Usage
Error handling
It is good practice to test the success/failure code after executing an INSERT statement.

Examples

**Example 1**
These statements insert a row with the values in EmpNbr and EmpName into the Emp_nbr and Emp_name columns of the Employee table identified in the default transaction object:

```
int EmpNbr
string EmpName
...
INSERT INTO Employee (employee.Emp_nbr,
    employee.Emp_name)
VALUES (:EmpNbr, :EmpName) ;
```

**Example 2**
These statements insert a row with the values entered in the SingleLineEdits sle_number and sle_name into the Emp_nbr and Emp_name columns of the Employee table in the transaction object named Emp_tran:

```
int EmpNbr
string EmpName
EmpNbr = Integer(sle_number.Text)
EmpName = sle_name.Text
INSERT INTO Employee (employee.Emp_nbr,
    employee.Emp_name)
VALUES (:EmpNbr, :EmpName) USING Emp_tran ;
```


**OPEN Cursor**

**Description**
Causes the SELECT specified when the cursor was declared to be executed.

**Syntax**

```
OPEN CursorName ;
```

**Parameter** | **Description**
--- | ---
CursorName | The name of the cursor you want to open

**Usage**
The USING TransactionObject clause is not allowed with OPEN; the transaction object was specified in the statement that declared the cursor.

**Error handling**
It is good practice to test the success/failure code after executing an OPEN Cursor statement.

**Examples**
This statement opens the cursor **Emp_curs**:

```
OPEN Emp_curs ;
```

**ROLLBACK**

**Description**
Cancels all database operations in the specified database since the last COMMIT, ROLLBACK, or CONNECT.

**Using COMMIT and ROLLBACK in a server component**
COMMIT and ROLLBACK commands embedded in a server component might have different effects depending on the setting of the UseContextObject DBParm parameter.

For information on the UseContextObject parameter see *Connecting to Your Database*. For information on deploying components to a transaction server, see *Application Techniques*.

**Syntax**

```
ROLLBACK {USING TransactionObject} ;
```

**Parameter** | **Description**
--- | ---
TransactionObject | The name of the transaction object that identifies the database in which you want to cancel all operations since the last COMMIT, ROLLBACK, or CONNECT. This clause is required only for transaction objects other than the default (SQLCA).
Usage

ROLLBACK does not cause a disconnect, but it does close all open cursors and procedures.

Error handling

It is good practice to test the success/failure code after executing a ROLLBACK statement.

Examples

Example 1  This statement cancels all database operations in the database specified in the default transaction object:

ROLLBACK ;

Example 2  This statement cancels all database operations in the database specified in the transaction object named Emp_tran:

ROLLBACK USING emp_tran ;

SELECT

Description

Selects a row in the tables specified in RestOfSelectStatement.

Syntax

SELECT RestOfSelectStatement
{USING TransactionObject} ;

Parameter | Description
--- | ---
RestOfSelectStatement | The rest of the SELECT statement (the column list INTO, FROM, WHERE, and other clauses).
TransactionObject | The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).

Usage

An error occurs if the SELECT statement returns more than one row.

Error handling

It is good practice to test the success/failure code after executing a SELECT statement. You can test SQLCode for a failure code.

When you use the INTO clause, PowerBuilder does not verify whether the datatype of the retrieved column matches the datatype of the host variable; it only checks for the existence of the columns and tables. You are responsible for checking that the datatypes match. Keep in mind that not all database datatypes are the same as PowerBuilder datatypes.
**SELECTBLOB**

**Examples**

The following statements select data in the Emp_LName and Emp_FName columns of a row in the Employee table and put the data into the SingleLineEdits sle_LName and sle_FName (the transaction object Emp_tran is used):

```powershell
int Emp_num
string Emp_lname, Emp_fname
Emp_num = Integer(sle_Emp_Num.Text)

SELECT employee.Emp_LName, employee.Emp_FName
    INTO :Emp_lname, :Emp_fname
    FROM Employee
    WHERE Employee.Emp_nbr = :Emp_num
    USING Emp_tran ;

IF Emp_tran.SQLCode = 100 THEN
    MessageBox("Employee Inquiry", & "Employee Not Found")
ELSEIF Emp_tran.SQLCode > 0 then
    MessageBox("Database Error", & Emp_tran.SQLErrText, Exclamation!)
END IF
sle_Lname.text = Emp_lname
sle_Fname.text = Emp_fname
```

**SELECTBLOB**

**Description**

Selects a single blob column in a row in the table specified in RestOfSelectStatement.

**Syntax**

```powershell
SELECTBLOB RestOfSelectStatement
    {USING TransactionObject} ;
```

**Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RestOfSelectStatement</td>
<td>The rest of the SELECT statement (the INTO, FROM, and WHERE clauses).</td>
</tr>
<tr>
<td>TransactionObject</td>
<td>The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).</td>
</tr>
</tbody>
</table>

**Usage**

An error occurs if the SELECTBLOB statement returns more than one row.
Error handling
It is good practice to test the success/failure code after executing an
SELECTBLOB statement. To make sure the update affected at least one row,
check the SQLNRows property of SQLCA or the transaction object. The
SQLCode or SQLDBCode property will not indicate the success or failure of
the SELECTBLOB statement.

You can include an indicator variable in the host variable list (target
parameters) in the INTO clause to check for an empty blob (a blob of zero
length) and conversion errors.

Database information
Sybase ASE and Microsoft SQL Server users must set the AutoCommit
property of the transaction object to true before calling the SELECTBLOB
function. For information about the AutoCommit property, see Connecting to
Your Database.

Examples
The following statements select the blob column Emp_pic from a row in the
Employee table and set the picture p_1 to the bitmap in Emp_id_pic (the
transaction object Emp_tran is used):

```powerscript
Blob Emp_id_pic
SELECTBLOB Emp_pic
    INTO :Emp_id_pic
    FROM Employee
    WHERE Employee.Emp_Num = 100
    USING Emp_tran ;
p_1.SetPicture(Emp_tran)
```

The blob Emp_id_pic requires a colon to indicate that it is a host (PowerScript)
variable when you use it in the INTO clause of the SELECTBLOB statement.
**UPDATE**

**Description**
Updates the rows specified in `RestOfUpdateStatement`.

**Syntax**
UPDATE `TableName` `RestOfUpdateStatement` (USING `TransactionObject`) ;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TableName</code></td>
<td>The name of the table in which you want to update rows.</td>
</tr>
<tr>
<td><code>RestOfUpdateStatement</code></td>
<td>The rest of the UPDATE statement (the SET and WHERE clauses).</td>
</tr>
<tr>
<td><code>TransactionObject</code></td>
<td>The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).</td>
</tr>
</tbody>
</table>

**Usage**

**Error handling**
It is good practice to test the success/failure code after executing a UPDATE statement. You can test SQLCode for a failure code. However, if nothing matches the WHERE clause and no rows are updated, SQLCode is still set to zero. To make sure the update affected at least one row, check the SQLNRows property of the transaction object.

**Examples**
These statements update rows from the Employee table in the database specified in the transaction object named `Emp_tran`, where `Emp_num` is equal to the value entered in the SingleLineEdit `sle_Number`:

```powerbuilder
int Emp_num
Emp_num=integer(sle_Number.Text)
UPDATE Employee
    SET emp_name = :sle_Name.Text
    WHERE Employee.emp_num = :Emp_num
    USING Emp_tran ;

IF Emptran.SQLNRows > 0 THEN
    COMMIT USING Emp_tran ;
END IF
```

The integer `Emp_num` and the SingleLineEdit `sle_name` require a colon to indicate they are host (PowerScript) variables when you use them in an UPDATE statement.
**UPDATEBLOB**

**Description**
Updates the rows in *TableName* in *BlobColumn*.

**Syntax**
```
UPDATEBLOB TableName
SET BlobColumn = BlobVariable
RestOfUpdateStatement {USING TransactionObject} ;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TableName</td>
<td>The name of the table you want to update.</td>
</tr>
<tr>
<td>BlobColumn</td>
<td>The name of the column you want to update in <em>TableName</em>. The datatype of this column must be blob.</td>
</tr>
<tr>
<td>BlobVariable</td>
<td>A PowerScript variable of the datatype blob.</td>
</tr>
<tr>
<td>RestOfUpdateStatement</td>
<td>The rest of the UPDATE statement (the WHERE clause).</td>
</tr>
<tr>
<td>TransactionObject</td>
<td>The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA).</td>
</tr>
</tbody>
</table>

**Usage**

**Error handling**
It is good practice to test the success/failure code after executing an UPDATEBLOB statement. To make sure the update affected at least one row, check the SQLNRows property of SQLCA or the transaction object. The SQLCode or SQLDBCode property will not indicate the success or failure of the UPDATEBLOB statement.

**Database information**
Sybase ASE and Microsoft SQL Server users must set the AutoCommit property of the transaction object to True before calling the UPDATEBLOB function. For information about the AutoCommit property, see *Connecting to Your Database*.

**Examples**
These statements update the blob column *emp_pic* in the *Employee* table, where *emp_num* is 100:
```
int fh
blob Emp_id_pic
fh = FileOpen("c:\emp_100.bmp", StreamMode!)
```
UPDATE Where Current of Cursor

Description
Updates the row in which the cursor is positioned using the values in SetStatement.

Syntax
UPDATE TableName SetStatement
WHERE CURRENT OF CursorName ;

Parameter | Description
--- | ---
TableName | The name of the table in which you want to update the row
SetStatement | The word SET followed by a comma-separated list of the form ColumnName = value
CursorName | The name of the cursor in which the table is referenced

Usage
The USING Transaction Object clause is not allowed with UPDATE Where Current of Cursor; the transaction object was specified in the statement that declared the cursor.

Examples
This statement updates the row in the Employee table in which the cursor called Emp_curs is positioned:

```
UPDATE Employee
SET salary = 17800
WHERE CURRENT of Emp_curs ;
```

The blob Emp_id_pic requires a colon to indicate it is a host (PowerScript) variable in the UPDATEBLOB statement.

IF fh <> -1 THEN
    FileRead(fh, emp_id_pic)
    FileClose(fh)
    UPDATEBLOB Employee SET emp_pic = :Emp_id_pic
    WHERE Emp_num = 100
    USING Emp_tran ;
END IF

IF Emptran.SQLNRows > 0 THEN
    COMMIT USING Emp_tran ;
END IF
Using dynamic SQL

General information

Because database applications usually perform a specific activity, you usually know the complete SQL statement when you write and compile the script. When PowerBuilder does not support the statement in embedded SQL (as with a DDL statement) or when the parameters or the format of the statements are unknown at compile time, the application must build the SQL statements at runtime. This is called dynamic SQL. The parameters used in dynamic SQL statements can change each time the program is executed.

Using SQL Anywhere

For information about using dynamic SQL with SQL Anywhere®, see the SQL Anywhere documentation.

Four formats

PowerBuilder has four dynamic SQL formats. Each format handles one of the following situations at compile time:

<table>
<thead>
<tr>
<th>Format</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format 1</td>
<td>Non-result-set statements with no input parameters</td>
</tr>
<tr>
<td>Format 2</td>
<td>Non-result-set statements with input parameters</td>
</tr>
<tr>
<td>Format 3</td>
<td>Result-set statements in which the input parameters and result-set columns are known at compile time</td>
</tr>
<tr>
<td>Format 4</td>
<td>Result-set statements in which the input parameters, the result-set columns or both are unknown at compile time</td>
</tr>
</tbody>
</table>

To handle these situations, you use:

- The PowerBuilder dynamic SQL statements
- The dynamic versions of CLOSE, DECLARE, FETCH, OPEN, and EXECUTE
- The PowerBuilder datatypes DynamicStagingArea and DynamicDescriptionArea

About the examples

The examples assume that the default transaction object (SQLCA) has been assigned valid values and that a successful CONNECT has been executed. Although the examples do not show error checking, you should check the SQLCode after each SQL statement.
Dynamic SQL statements

The PowerBuilder dynamic SQL statements are:

```
DESCRIBE DynamicStagingArea
  INTO DynamicDescriptionArea ;
EXECUTE (IMMEDIATE) SQLStatement
  {USING TransactionObject} ;
EXECUTE DynamicStagingArea
  USING ParameterList ;
EXECUTE DYNAMIC Cursor | Procedure
  USING ParameterList ;
OPEN DYNAMIC Cursor | Procedure
  USING ParameterList ;
EXECUTE DYNAMIC Cursor | Procedure
  USING DESCRIPTOR DynamicDescriptionArea ;
OPEN DYNAMIC Cursor | Procedure
  USING DESCRIPTOR DynamicDescriptionArea ;
PREPARE DynamicStagingArea
  FROM SQLStatement {USING TransactionObject} ;
```

Two datatypes

**DynamicStagingArea**  DynamicStagingArea is a PowerBuilder datatype. PowerBuilder uses a variable of this type to store information for use in subsequent statements.

The DynamicStagingArea is the only connection between the execution of a statement and a transaction object and is used internally by PowerBuilder; you cannot access information in the DynamicStagingArea.

PowerBuilder provides a global DynamicStagingArea variable named SQLSA that you can use when you need a DynamicStagingArea variable.

If necessary, you can declare and create additional object variables of the type DynamicStagingArea. These statements declare and create the variable, which must be done before referring to it in a dynamic SQL statement:

```
DynamicStagingArea dsa_stage1
  dsa_stage1 = CREATE DynamicStagingArea
```

After the EXECUTE statement is completed, SQLSA is no longer referenced.

**DynamicDescriptionArea**  DynamicDescriptionArea is a PowerBuilder datatype. PowerBuilder uses a variable of this type to store information about the input and output parameters used in Format 4 of dynamic SQL.

PowerBuilder provides a global DynamicDescriptionArea named SQLDA that you can use when you need a DynamicDescriptionArea variable.
If necessary, you can declare and create additional object variables of the type DynamicDescriptionArea. These statements declare and create the variable, which must be done before referring to it in a dynamic SQL statement:

```
DynamicDescriptionArea dda_desc1
dsa_desc1 = CREATE DynamicDescriptionArea
```

For more information about SQLDA, see Dynamic SQL Format 4 on page 176.

When you use dynamic SQL, you must:

- Prepare the DynamicStagingArea in all formats except Format 1
- Describe the DynamicDescriptionArea in Format 4
- Execute the statements in the appropriate order

**Preparing and describing the datatypes** Since the SQLSA staging area is the only connection between the execution of a SQL statement and a transaction object, an execution error will occur if you do not prepare the SQL statement correctly.

In addition to SQLSA and SQLDA, you can declare other variables of the DynamicStagingArea and DynamicDescriptionArea datatypes. However, this is required only when your script requires simultaneous access to two or more dynamically prepared statements.

This is a *valid* dynamic cursor:

```
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
PREPARE SQLSA FROM "SELECT emp_id FROM employee" ;
OPEN DYNAMIC my_cursor ;
```

This is an *invalid* dynamic cursor. There is no PREPARE, and therefore an execution error will occur:

```
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
OPEN DYNAMIC my_cursor ;
```

**Statement order** Where you place the dynamic SQL statements in your scripts is unimportant, but the order of execution is important in Formats 2, 3, and 4. You must execute:

1. The DECLARE and the PREPARE before you execute any other dynamic SQL statements
2. The OPEN in Formats 3 and 4 before the FETCH
3. The CLOSE at the end
If you have multiple PREPARE statements, the order affects the contents of SQLSA.

These statements illustrate the correct ordering:

```powershell
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA
string sql1, sql2
sql1 = "SELECT emp_id FROM department " &
WHERE salary > 90000"
sql2 = "SELECT emp_id FROM department " &
WHERE salary > 20000"

IF deptId = 200 then
    PREPARE SQLSA FROM :sql1 USING SQLCA ;
ELSE
    PREPARE SQLSA FROM :sql2 USING SQLCA ;
END IF
OPEN DYNAMIC my_cursor ; // my_cursor maps to the
// SELECT that has been
// prepared.
```

When you connect to Microsoft SQL Server using the PowerBuilder SQL Native Client (SNC) database interface, the syntax for declaring a procedure is:

```powershell
DECLARE logical_procedure_name PROCEDURE FOR
[@rc=]procedure_name
[@param1 = value1 [OUTPUT], @param2 = value2 [OUTPUT], ...]
[USING transaction_object];
```

[@rc=] indicates that you want to get the procedure’s return value.

Use the keyword OUTPUT or OUT to indicate an output parameter if you want to get the output parameter’s value.

If the BindSPInput database parameter is 0, `value1, value2,...` can be either PowerBuilder script variables or literal values. If BindSPInput is 1, `value1, value2,...` must be PowerBuilder script variables. If you specify literal values, the SNC interface returns a runtime error.

When you declare a dynamic SQL statement with a procedure, enter a question mark (?) for each IN/OUT parameter in the statement. Value substitution is positional. For examples, see Dynamic SQL Format 3 and 4.
Dynamic SQL Format 1

Description
Use this format to execute a SQL statement that does not produce a result set and does not require input parameters. You can use this format to execute all forms of Data Definition Language (DDL).

Syntax
EXECUTE IMMEDIATE SQLStatement {USING TransactionObject} ;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLStatement</td>
<td>A string containing a valid SQL statement. The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions.</td>
</tr>
<tr>
<td>TransactionObject (optional)</td>
<td>The name of the transaction object that identifies the database.</td>
</tr>
</tbody>
</table>

Examples
These statements create a database table named Trainees. The statements use the string Mysql to store the CREATE statement.

For Sybase ASE and Microsoft SQL Server users
If you are connected to an ASE or SQL Server database, set AUTOCOMMIT to true before executing the CREATE.

```powerbuilder
string Mysql
Mysql = "CREATE TABLE Trainees "&
   +"(emp_id integer not null,”&
    +"emp_fname char(10) not null, "&
    +"emp_lname char(20) not null)"
EXECUTE IMMEDIATE :Mysql ;
```

These statements assume a transaction object named My_trans exists and is connected:

```powerbuilder
string Mysql
Mysql="INSERT INTO department Values (1234,"&
   +"'Purchasing',1234)"
EXECUTE IMMEDIATE :Mysql USING My_trans ;
```
Dynamic SQL Format 2

Description
Use this format to execute a SQL statement that does not produce a result set but does require input parameters. You can use this format to execute all forms of Data Definition Language (DDL).

Syntax
PREPARE DynamicStagingArea FROM SQLStatement
    (USING TransactionObject) ;
EXECUTE DynamicStagingArea USING {ParameterList} ;

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicStagingArea</td>
<td>The name of the DynamicStagingArea (usually SQLSA).</td>
</tr>
<tr>
<td></td>
<td>If you need a DynamicStagingArea variable other than SQLSA, you must declare it and instantiate it with the CREATE statement before using it.</td>
</tr>
<tr>
<td>SQLStatement</td>
<td>A string containing a valid SQL statement. The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions. Enter a question mark (?) for each parameter in the statement. Value substitution is positional; reserved word substitution is not allowed.</td>
</tr>
<tr>
<td>TransactionObject</td>
<td>(optional) The name of the transaction object that identifies the database.</td>
</tr>
<tr>
<td>ParameterList</td>
<td>(optional) A comma-separated list of PowerScript variables. Note that PowerScript variables are preceded by a colon (:)</td>
</tr>
</tbody>
</table>

Usage
To specify a null value, use the SetNull function.

Examples
These statements prepare a DELETE statement with one parameter in SQLSA and then execute it using the value of the PowerScript variable Emp_id_var:

```
INT Emp_id_var = 56
PREPARE SQLSA
    FROM "DELETE FROM employee WHERE emp_id=?" ;
EXECUTE SQLSA USING :Emp_id_var ;
```

These statements prepare an INSERT statement with three parameters in SQLSA and then execute it using the value of the PowerScript variables Dept_id_var, Dept_name_var, and Mgr_id_var (note that Mgr_id_var is null):

```
INT Dept_id_var = 156
INT Mgr_id_var
String Dept_name_var
Dept_name_var = "Department"
SetNull(Mgr_id_var)
```
PREPARE SQLSA
   FROM "INSERT INTO department VALUES (?,?,?)" ;
EXECUTE SQLSA
   USING :Dept_id_var,:Dept_name_var,:Mgr_id_var ;

Dynamic SQL Format 3

Description
Use this format to execute a SQL statement that produces a result set in which
the input parameters and result set columns are known at compile time.

Syntax
DECLARE Cursor | Procedure
   DYNAMIC CURSOR | PROCEDURE
   FOR DynamicStagingArea ;
PREPARE DynamicStagingArea FROM SQLStatement
   {USING TransactionObject} ;
OPEN DYNAMIC Cursor
   {USING ParameterList} ;
EXECUTE DYNAMIC Procedure
   {USING ParameterList} ;
FETCH Cursor | Procedure
   INTO HostVariableList ;
CLOSE Cursor | Procedure ;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor or Procedure</td>
<td>The name of the cursor or procedure you want to use.</td>
</tr>
<tr>
<td>DynamicStagingArea</td>
<td>The name of the DynamicStagingArea (usually SQLSA).</td>
</tr>
<tr>
<td></td>
<td>If you need a DynamicStagingArea variable other than SQLSA, you must declare it and instantiate it with the CREATE statement before using it.</td>
</tr>
<tr>
<td>SQLStatement</td>
<td>A string containing a valid SQL SELECT statement</td>
</tr>
<tr>
<td></td>
<td>The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions.</td>
</tr>
<tr>
<td></td>
<td>Enter a question mark (?) for each parameter in the statement. Value substitution is positional; reserved word substitution is not allowed.</td>
</tr>
<tr>
<td>TransactionObject</td>
<td>The name of the transaction object that identifies the database.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
</tr>
</tbody>
</table>
Usage

To specify a null value, use the SetNull function.

The DECLARE statement is not executable and can be declared globally.

If your DBMS supports formats of FETCH other than the customary (and default) FETCH NEXT, you can specify FETCH FIRST, FETCH PRIOR, or FETCH LAST.

The FETCH and CLOSE statements in Format 3 are the same as in standard embedded SQL.

To declare a local cursor or procedure, open the script in the Script view and select Paste SQL from the PainterBar or the Edit>Paste Special menu. To declare a global, instance, or shared cursor or procedure, select Declare from the first drop-down list in the Script view, and select Global Variables, Instance Variables, or Shared Variables from the second drop-down list. Then, select Paste SQL.

For information about global, instance, shared, and local scope, see “Where to declare variables” on page 34.

Examples

Example 1  These statements associate a cursor named my_cursor with SQLSA, prepare a SELECT statement in SQLSA, open the cursor, and return the employee ID in the current row into the PowerScript variable Emp_id_var:

```powerbuilder
integer Emp_id_var
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
PREPARE SQLSA FROM "SELECT emp_id FROM employee" ;
OPEN DYNAMIC my_cursor ;
FETCH my_cursor INTO :Emp_id_var ;
CLOSE my_cursor ;
```

You can loop through the cursor as you can in embedded static SQL.

Example 2  These statements associate a cursor named my_cursor with SQLSA, prepare a SELECT statement with one parameter in SQLSA, open the cursor, and substitute the value of the variable Emp_state_var for the parameter in the SELECT statement. The employee ID in the active row is returned into the PowerBuilder variable Emp_id_var:

```powerbuilder
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
integer Emp_id_var
```
string Emp_state_var = "MA"
string sqlstatement

sqlstatement = "SELECT emp_id FROM employee " & +"WHERE state = ?"
PREPARE SQLSA FROM :sqlstatement ;
OPEN DYNAMIC my_cursor using :Emp_state_var ;
FETCH my_cursor INTO :Emp_id_var ;
CLOSE my_cursor ;

Example 3  These statements perform the same processing as the preceding example but use a database stored procedure called Emp_select:

// The syntax of emp_select is:
// create procedure emp_select (@stateparm char(2)) as
// SELECT emp_id FROM employee WHERE state=@stateparm.
DECLARE my_proc DYNAMIC PROCEDURE FOR SQLSA ;
integer Emp_id_var
string Emp_state_var

PREPARE SQLSA FROM "execute emp_select @stateparm=?" ;
Emp_state_var = "MA"
EXECUTE DYNAMIC my_proc USING :Emp_state_var ;
FETCH my_proc INTO :Emp_id_var ;
CLOSE my_proc ;

Example 4  These statements are for a stored procedure with a return value for a SQL Native Client (SNC) connection:

integer var1, ReturnVal
string var2

PREPARE SQLSA FROM "execute @rc = myproc @parm1=?, @parm2=? OUTPUT ";
DECLARE my_proc DYNAMIC PROCEDURE FOR SQLSA ;

EXECUTE DYNAMIC my_proc USING :var1, :var2 ;

//fetch result set
...

//fetch return value and output parameter
FETCH my_proc INTO : ReturnVal, :var2;
CLOSE my_proc ;
Dynamic SQL Format 4

Description
Use this format to execute a SQL statement that produces a result set in which the number of input parameters, or the number of result-set columns, or both, are unknown at compile time.

Syntax
DECLARE Cursor | Procedure
    DYNAMIC CURSOR | PROCEDURE
    FOR DynamicStagingArea ;
PREPARE DynamicStagingArea FROM SQLStatement
    {USING TransactionObject} ;
DESCRIBE DynamicStagingArea
    INTO DynamicDescriptionArea ;
OPEN DYNAMIC Cursor
    USING DESCRIPTOR DynamicDescriptionArea ;
EXECUTE DYNAMIC Procedure
    USING DESCRIPTOR DynamicDescriptionArea ;
FETCH Cursor | Procedure
    USING DESCRIPTOR DynamicDescriptionArea ;
CLOSE Cursor | Procedure ;

Parameter | Description
--- | ---
Cursor or Procedure | The name of the cursor or procedure you want to use.
DynamicStagingArea | The name of the DynamicStagingArea (usually SQLSA).
    If you need a DynamicStagingArea variable other than SQLSA, you must declare it and instantiate it with the CREATE statement before using it.
SQLStatement | A string containing a valid SQL SELECT statement. The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions.
    Enter a question mark (?) for each parameter in the statement. Value substitution is positional; reserved word substitution is not allowed.
TransactionObject (optional) | The name of the transaction object that identifies the database.
DynamicDescriptionArea | The name of the DynamicDescriptionArea (usually SQLDA).
    If you need a DynamicDescriptionArea variable other than SQLDA, you must declare it and instantiate it with the CREATE statement before using it.
CHAPTER 8    SQL Statements

Usage

The DECLARE statement is not executable and can be defined globally.

If your DBMS supports formats of FETCH other than the customary (and default) FETCH NEXT, you can specify FETCH FIRST, FETCH PRIOR, or FETCH LAST.

To declare a local cursor or procedure, open the script in the Script view and select Paste SQL from the PainterBar or the Edit>Paste Special menu. To declare a global, instance, or shared cursor or procedure, select Declare from the first drop-down list in the Script view and Global Variables, Instance Variables, or Shared Variables from the second drop-down list, then select Paste SQL.

For information about global, instance, shared, and local scope, see “Where to declare variables” on page 34.

Accessing attribute information

When a statement is described into a DynamicDescriptionArea, this information is available to you in the attributes of that DynamicDescriptionArea variable:

<table>
<thead>
<tr>
<th>Information</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input parameters</td>
<td>NumInputs</td>
</tr>
<tr>
<td>Array of input parameter types</td>
<td>InParmType</td>
</tr>
<tr>
<td>Number of output parameters</td>
<td>NumOutputs</td>
</tr>
<tr>
<td>Array of output parameter types</td>
<td>OutParmType</td>
</tr>
</tbody>
</table>

Setting and accessing parameter values

The array of input parameter values and the array of output parameter values are also available. You can use the SetDynamicParm function to set the values of an input parameter and the following functions to obtain the value of an output parameter:

- GetDynamicDate
- GetDynamicDateTime
- GetDynamicDecimal
- GetDynamicNumber
- GetDynamicString
- GetDynamicTime

For information about these functions, see GetDynamicDate on page 544, GetDynamicDateTime on page 546, GetDynamicDecimal on page 548, GetDynamicNumber on page 549, GetDynamicString on page 550, and GetDynamicTime on page 551.
Parameter values  The following enumerated datatypes are the valid values for the input and output parameter types:

- TypeBoolean!
- TypeByte!
- TypeDate!
- TypeDateTime!
- TypeDecimal!
- TypeDouble!
- TypeInteger!
- TypeLong!
- TypeLongLong!
- TypeReal!
- TypeString!
- TypeTime!
- TypeUInt!
- TypeULong!
- TypeUnknown!

Input parameters  You can set the type and value of each input parameter found in the PREPARE statement. PowerBuilder populates the SQLDA attribute NumInputs when the DESCRIBE is executed. You can use this value with the SetDynamicParm function to set the type and value of a specific input parameter. The input parameters are optional; but if you use them, you should fill in all the values before executing the OPEN or EXECUTE statement.

Output parameters  You can access the type and value of each output parameter found in the PREPARE statement. If the database supports output parameter description, PowerBuilder populates the SQLDA attribute NumOutputs when the DESCRIBE is executed. If the database does not support output parameter description, PowerBuilder populates the SQLDA attribute NumOutputs when the FETCH statement is executed.

You can use the number of output parameters in the NumOutputs attribute in functions to obtain the type of a specific parameter from the output parameter type array in the OutParmType attribute. When you have the type, you can call the appropriate function after the FETCH statement to retrieve the output value.

Examples

Example 1  These statements assume you know that there will be only one output descriptor and that it will be an integer. You can expand this example to support any number of output descriptors and any datatype by wrapping the CHOOSE CASE statement in a loop and expanding the CASE statements:

```plaintext
string Stringvar, Sqlstatement
integer Intvar
Long LongVar
```
 Sqlstatement = "SELECT emp_id FROM employee"
 PREPARE SQLSA FROM :Sqlstatement ;
 DESCRIBE SQLSA INTO SQLDA ;
 DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
 OPEN DYNAMIC my_cursor USING DESCRIPTOR SQLDA ;
 FETCH my_cursor USING DESCRIPTOR SQLDA ;
 // If the FETCH is successful, the output
 // descriptor array will contain returned
 // values from the first row of the result set.
 // SQLDA.NumOutputs contains the number of
 // output descriptors.
 // The SQLDA.OutParmType array will contain
 // NumOutput entries and each entry will contain
 // a value of the enumerated datatype ParmType
 // (such as TypeInteger!, TypeLongLong!, or
 // TypeString!).
 CHOOSE CASE SQLDA.OutParmType[1]
   CASE TypeString!
     Stringvar = GetDynamicString(SQLDA, 1)
   CASE TypeInteger!
     Intvar = GetDynamicNumber(SQLDA, 1)
   CASE TypeLongLong!
     Longvar = GetDynamicDecimal(SQLDA, 1)
 END CHOOSE
 CLOSE my_cursor ;

 Example 2  These statements assume you know there is one string input
descriptor and sets the parameter to MA:

 string Sqlstatement, sValue
 Sqlstatement = "SELECT emp_fname, emp_lname " &
 + "FROM employee WHERE state = ?"
 PREPARE SQLSA FROM :Sqlstatement ;
 DESCRIBE SQLSA INTO SQLDA ;

 // If the DESCRIBE is successful, the input
 // descriptor array will contain one input
 // descriptor that you must fill prior to the OPEN

 DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
 SetDynamicParm(SQLDA, 1, "MA")

 OPEN DYNAMIC my_cursor USING DESCRIPTOR SQLDA ;

 FETCH my_cursor USING DESCRIPTOR SQLDA ;

 }
// If the FETCH is successful, the output
// descriptor array will contain returned
// values from the first row of the result set
// as in the first example.

// To test and see the values:
sValue = SQLDA.GetDynamicString(1)
//messagebox("",sValue)
sValue = SQLDA.GetDynamicString(2)
//messagebox("",sValue)
Do While sqlca.sqlcode <> 100
  FETCH my_cursor USING DESCRIPTOR SQLDA ;
  sValue = SQLDA.GetDynamicString(1)
  //messagebox("",sValue)
  sValue = SQLDA.GetDynamicString(2)
  //messagebox("",sValue)
Loop
CLOSE my_cursor ;

**Example 3**  This example is for a stored procedure with a return value for a SQL Native Client (SNC) connection:

```powerbuilder
integer var1, ReturnVal
string var2

PREPARE SQLSA FROM  "execute @rc = myproc @parm1=?,
                   @parm2=? OUTPUT ";

DESCRIBE SQLSA INTO SQLDA ;

DECLARE my_proc DYNAMIC PROCEDURE FOR SQLSA ;

SetDynamicParm(SQLDA, 1, var1)
SetDynamicParm(SQLDA, 2, var2)

EXECUTE DYNAMIC my_proc USING DESCRIPTOR SQLDA ;

//fetch result set
...

//fetch return value and output parameter
FETCH my_proc USING DESCRIPTOR SQLDA ;

//get return value
CHOOSE CASE SQLDA.OutParmType[1]
  CASE TypeInteger!
```
rc = GetDynamicNumber(SQLDA, 1)
CASE TypeLong!
  rc = GetDynamicNumber(SQLDA, 1)
END CASE
CASE TypeString!
  Var2 = GetDynamicString(SQLDA, 1)
END CASE

//get output value

CHOOSE CASE SQLDA.OutParmType[2]
CASE TypeString!
  Var2 = GetDynamicString(SQLDA, 2)
CASE TypeInteger!
  rc = GetDynamicNumber(SQLDA, 2)
CASE TypeLong!
  rc = GetDynamicNumber(SQLDA, 2)
END CASE

CLOSE my_proc ;
About this chapter

This chapter discusses events in general and then documents the arguments, event IDs, and return codes for the events defined for all PowerBuilder controls and objects except the DataWindow and DataStore. Usage notes and examples provide information about what is typically done in an event’s script.

For information about DataWindow and DataStore events, see the DataWindow Reference.

Contents

The events are listed in alphabetical order.

About events

In PowerBuilder, there are several types of events.

Table 9-1: PowerBuilder event types

<table>
<thead>
<tr>
<th>Type</th>
<th>Occurs in response to</th>
</tr>
</thead>
<tbody>
<tr>
<td>System events with an ID</td>
<td>User actions or other system messages or a call in your scripts</td>
</tr>
<tr>
<td>System events without an ID</td>
<td>PowerBuilder messages or a call in your scripts</td>
</tr>
<tr>
<td>User-defined events with an ID</td>
<td>User actions or other system messages or a call in your scripts</td>
</tr>
<tr>
<td>User-defined events without an ID</td>
<td>A call in your scripts</td>
</tr>
</tbody>
</table>

The following information about event IDs, arguments, and return values applies to all types of events.

Event IDs

An event ID connects an event to a system message. Events that can be triggered by user actions or other system activity have event IDs. In PowerBuilder’s objects, PowerBuilder defines events for commonly used event IDs. These events are documented in this chapter. You can define your own events for other system messages using the event IDs listed in the Event Declaration dialog box.


About events

Events without IDs  Some system events, such as the application object’s Open event, do not have an event ID. They are associated with PowerBuilder activity, not system activity. PowerBuilder triggers them itself when appropriate.

Arguments

System-triggered events  Each system event has its own list of zero or more arguments. When PowerBuilder triggers the event in response to a system message, it supplies values for the arguments, which become available in the event script.

Events you trigger  If you trigger a system event in another event script, you specify the expected arguments. For example, in the Clicked event for a window, you can trigger the DoubleClicked event with this statement, passing its flags, xpos, and ypos arguments on to the DoubleClicked event.

```powershell
w_main.EVENT DoubleClicked(flags, xpos, ypos)
```

Because DoubleClicked is a system event, the argument list is fixed—you cannot supply additional arguments of your own.

Calling events without specifying their arguments

If you use the CALL statement, you can trigger a system event without specifying its arguments. However, CALL is obsolete and you should not use it in new applications except as described in CALL on page 123.

Return values

Where does the return value go?  Most events have a return value. When the event is triggered by the system, the return value is returned to the system.

When your script triggers a user-defined or system event, you can capture the return value in an assignment statement:

```powershell
li_rtn = w_main.EVENT process_info(mydata)
```

When you post an event, the return value is lost because the calling script is no longer running when the posted script is actually run. The compiler does not allow a posted event in an assignment statement.

Return codes  System events with return values have a default return code of 0, which means, “take no special action and continue processing.” Some events have additional codes that you can return to change the processing that happens after the event. For example, a return code might allow you to suppress an error message or prevent a change from taking place.

A RETURN statement is not required in an event script, but for most events it is good practice to include one. For events with return values, if you do not have a RETURN statement, the event returns 0.
Some system events have no return value. For these events, the compiler does not allow a RETURN statement.

Sometimes you want to perform some processing in an event in a descendent object, but that processing depends on the return value of the ancestor event script. You can use a local variable called AncestorReturnValue that is automatically declared and assigned the value of the ancestor event.

For more information about AncestorReturnValue, see “Calling functions and events in an object’s ancestor” on page 114.

### User-defined events

**With an ID** When you declare a user-defined event that will be triggered by a system message, you select an event ID from the list of IDs. The pbm (PowerBuilder Message) codes listed in the Event dialog box map to system messages.

The return value and arguments associated with the event ID become part of your event declaration. You cannot modify them.

When the corresponding system message occurs, PowerBuilder triggers the event and passes values for the arguments to the event script.

**Without an ID** When you declare a user event that will not be associated with a system message, you do not select an event ID for the event.

You can specify your own arguments and return datatype in the Event Declaration dialog box.

The event will never be triggered by user actions or system activity. You trigger the event yourself in your application’s scripts.

If you want to trigger events, including system events, see “Syntax for calling PowerBuilder functions and events” on page 111 for information on the calling syntax.

To learn more about user-defined events, see the PowerBuilder Users Guide.
Activate

Description
Occurs just before the window becomes active.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_activate</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

Usage
When an Activate event occurs, the first object in the tab order for the window gets focus. If there are no visible objects in the window, the window gets focus.

An Activate event occurs for a newly opened window because it is made active after it is opened.

The Activate event is frequently used to enable and disable menu items.

Examples

**Example 1**  In the window’s Activate event, this code disables the Sheet menu item for menu m_frame on the File menu:

```powerbuilder
m_frame.m_file.m_sheet.Enabled = FALSE
```

**Example 2**  This code opens the sheet w_sheet in a layered style when the window activates:

```powerbuilder
w_sheet.ArrangeSheets(Layer!)
```

See also
Close
Open
Show

BeginDownload

Description
Occurs at the beginning of a download procedure

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
None
Usage

Use this event to add custom actions at the beginning of the download stage of a synchronization.

When the MobiLink synchronization server receives data, it updates the consolidated database, then builds a download stream that contains all relevant changes and sends it back to the remote site. At the end of each successful synchronization, the consolidated and remote databases are consistent. Either a whole transaction is synchronized, or none of it is synchronized. This ensures transactional integrity at each database.

The BeginDownload event marks the beginning of the download transaction. For a complete list of connection and synchronization events, and examples of their use, see the MobiLink documentation.

See also

BeginSync
BeginUpload
ConnectMobiLink

BeginDrag

The BeginDrag event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

For ListView controls

Description

Occurs when the user presses the left mouse button in the ListView control and begins dragging.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnbegindrag</td>
<td>ListView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value (the index of the ListView item being dragged)</td>
</tr>
</tbody>
</table>
### BeginDrag

**Return codes**

<table>
<thead>
<tr>
<th>Long. Return code choices (specify in a RETURN statement):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  Continue processing</td>
</tr>
</tbody>
</table>

**Usage**

BeginDrag and BeginRightDrag events occur when the user presses the mouse button and drags, whether or not dragging is enabled. To enable dragging, you can:

- Set the DragAuto property to true. If the ListView’s DragAuto property is true, a drag operation begins automatically when the user clicks.

- Call the Drag function. If DragAuto is false, then in the BeginDrag event script, the programmer can call the Drag function to begin the drag operation.

Dragging a ListView item onto another control causes its standard drag events (DragDrop, DragEnter, DragLeave, and DragWithin) to occur. The standard drag events occur for ListView when another control is dragged within the borders of the ListView.

**Examples**

This example moves a ListView item from one ListView to another. Ilvi_dragged_object is a window instance variable whose type is ListViewItem. To copy the item, omit the code that deletes it from the source ListView.

This code is in the BeginDrag event script of the source ListView:

```powerbuilder
// If the TreeView's DragAuto property is FALSE
This.Drag(Begin!)

This.GetItem(This.SelectedIndex(), & ilvi_dragged_object)

// To copy, rather than move, omit these two lines
This.DeleteItem(This.SelectedIndex())
This.Arrange()
```

This code is in the DragDrop event of the target ListView:

```powerbuilder
This.AddItem(ilvi_dragged_object)
This.Arrange()
```

**See also**

BeginRightDrag
DragDrop
DragEnter
DragLeave
DragWithin
Syntax 2  
For TreeView controls

Description
Occurs when the user presses the left mouse button on a label in the TreeView control and begins dragging.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnbegindrag</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by value (handle of the TreeView item being dragged)</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):
0  Continue processing

Usage
BeginDrag and BeginRightDrag events occur when the user presses the mouse button and drags, whether or not dragging is enabled. To enable dragging, you can:

- Set the DragAuto property to true. If the TreeView’s DragAuto property is true, a drag operation begins automatically when the user clicks.
- Call the Drag function. If DragAuto is false, then in the BeginDrag event script, the programmer can call the Drag function to begin the drag operation.

The user cannot drag a highlighted item.

Dragging a TreeView item onto another control causes the control’s standard drag events (DragDrop, DragEnter, DragLeave, and DragWithin) to occur. The standard drag events occur for TreeView when another control is dragged within the borders of the TreeView.

Examples
This example moves the first TreeView item in the source TreeView to another TreeView when the user drags there. Itvi_dragged_object is a window instance variable whose type is TreeViewItem. To copy the item, omit the code that deletes it from the source TreeView.

This code is in the BeginDrag event script of the source TreeView:

```powerscript
long itemnum

// If the TreeView's DragAuto property is FALSE
This.Drag(Begin!)
itemnum = 1
This.GetItem(itemnum, itvi_dragged_object)
```
// To copy, rather than move, omit these two lines
This.DeleteItem(itemnum)
This.SetRedraw(TRUE)

This code is in the DragDrop event of the target TreeView:

This.InsertItemLast(0, ilvi_dragged_object)
This.SetRedraw(TRUE)

Instead of deleting the item from the source TreeView immediately, consider deleting it after the insertion in the DragDrop event succeeds.

See also
- BeginRightDrag
- DragDrop
- DragEnter
- DragLeave
- DragWithin

**BeginLabelEdit**

The BeginLabelEdit event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For ListView controls**

Occurs when the user clicks on the label of an item after selecting the item.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnbeginlabeledit</td>
<td>ListView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value (the index of the selected ListView item)</td>
</tr>
</tbody>
</table>
Return codes  

Long. Return code choices (specify in a RETURN statement):

- 0  Allow editing of the label
- 1  Prevent editing of the label

Usage

When editing is allowed, a box appears around the label with the text highlighted. The user can replace or change the existing text.

Examples

This example uses the BeginLabelEdit event to display the name of the ListView item being edited:

```powerscript
ListViewItem lvi
This.GetItem(index lvi)
sle_info.text = "Editing " + string(lvi.label)
```

See also

EndLabelEdit

---

Syntax 2

For TreeView controls

Description

Occurs when the user clicks on the label of an item after selecting the item.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnbeginlabeledit</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by value (the handle of the selected TreeView item)</td>
</tr>
</tbody>
</table>

Return codes  

Long. Return code choices (specify in a RETURN statement):

- 0  Allow editing of the label
- 1  Prevent editing of the label

Usage

When editing is allowed, a box appears around the label with the text highlighted. The user can replace or change the existing text.

Examples

This example uses the BeginLabelEdit to display the name of the TreeView item being edited in a SingleLineEdit:

```powerscript
TreeViewItem tvi
This.GetItem(index, tvi)
sle_info.text = "Editing " + string(tvi.label)
```

See also

EndLabelEdit
**BeginLogScan**

**Description**
Occurs before `dbmlsync` scans the transaction log to assemble the upload data stream.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSync</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rescanlog</code></td>
<td>Boolean indicating whether the log has already been scanned for the current synchronization.</td>
</tr>
</tbody>
</table>

**Return codes**
None

**Usage**
Use this event to add custom actions immediately before the transaction log is scanned for upload. The following events are triggered while the upload stream is prepared, but before synchronization begins: `BeginLogScan`, `ProgressInfo`, and `EndLogScan`.

If this is the first time the transaction log has been scanned for this synchronization, the `rescanlog` value is false; otherwise it is true. The log is scanned twice when the MobiLink synchronization server and `dbmlsync` have different information about where scanning should begin.

**See also**

- `EndLogScan`
- `ProgressIndex`

---

**BeginRightDrag**

The `BeginRightDrag` event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>
CHAPTER 9  PowerScript Events

Syntax 1  For ListView controls

Description
Occurs when the user presses the right mouse button in the ListView control and begins dragging.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnbeginrightdrag</td>
<td>ListView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value (the index of the ListView item being dragged)</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):
- 0  Continue processing

Usage
BeginDrag and BeginRightDrag events occur when the user presses the mouse button and drags, whether or not dragging is enabled. To enable dragging, you can:
- Set the DragAuto property to true. If the ListView’s DragAuto property is true, a drag operation begins automatically when the user clicks.
- Call the Drag function. If DragAuto is false, then in the BeginRightDrag event script, the programmer can call the Drag function to begin the drag operation.

Dragging a ListView item onto another control causes its standard drag events (DragDrop, DragEnter, DragLeave, and DragWithin) to occur. The standard drag events occur for ListView when another control is dragged within the borders of the ListView.

Examples
See the example for the BeginDrag event. It is also effective for the BeginRightDrag event.

See also
BeginDrag
DragDrop
DragEnter
DragLeave
DragWithin
**BeginRightDrag**

**Syntax 2**

**For TreeView controls**

Description
Occurs when the user presses the right mouse button in the TreeView control and begins dragging.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbn_tvnbeginrightdrag</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by value (the handle of the TreeView item being dragged)</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

Usage
BeginDrag and BeginRightDrag events occur when the user presses the mouse button and drags, whether or not dragging is enabled. To enable dragging, you can:

- Set the DragAuto property to true. If the ListView’s DragAuto property is true, a drag operation begins automatically when the user clicks.

- Call the Drag function. If DragAuto is false, then in the BeginRightDrag event script, the programmer can call the Drag function to begin the drag operation.

The user cannot drag a highlighted item. Dragging a TreeView item onto another control causes its standard drag events (DragDrop, DragEnter, DragLeave, and DragWithin) to occur. The standard drag events occur for TreeView when another control is dragged within the borders of the TreeView.

Examples
See the example for the BeginDrag event.

See also
BeginDrag
DragDrop
DragEnter
DragLeave
DragWithin
CHAPTER 9  PowerScript Events

BeginSync

Description
Occurs at the beginning of the synchronization.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlusername</td>
<td>Read-only string identifying the MobiLink user name.</td>
</tr>
<tr>
<td>pubnames</td>
<td>Read-only string identifying the publication to be synchronized. If there is more than one publication, this is a comma-separated list.</td>
</tr>
</tbody>
</table>

Return codes
None

Usage
Use this event to add custom actions at the beginning of a synchronization. The following synchronization object events correspond to events occurring on the synchronization server (in the order displayed): BeginSync, ConnectMobiLink, BeginUpload, EndUpload, BeginDownload, EndDownload, DisconnectMobiLink, and EndSync.

See also
BeginDownload
BeginUpload
ConnectMobiLink

BeginUpload

Description
Occurs at the beginning of the synchronization upload procedure.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
None

Usage
Use this event to add custom actions immediately before the transmission of the upload to the MobiLink synchronization server.
The BeginUpload event marks the beginning of the upload transaction. Applicable inserts and updates to the consolidated database are performed for all remote tables, then rows are deleted as applicable for all remote tables. After EndUpload, upload changes are committed.

See also
BeginDownload
ConnectMobiLink
EndUpload

**Clicked**

The Clicked event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menus</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>ListView and Toolbar controls</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Tab controls</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>TreeView controls</td>
<td>Syntax 4</td>
</tr>
<tr>
<td>Window and progress bar controls</td>
<td>Syntax 5</td>
</tr>
<tr>
<td>Other controls</td>
<td>Syntax 6</td>
</tr>
</tbody>
</table>

For information about the DataWindow control's Clicked event, see the DataWindow Reference or the online Help.

### Syntax 1

**For menus**

Occurs when the user chooses an item on a menu.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Menu</td>
</tr>
</tbody>
</table>

Arguments

None

Return codes

None (do not use a RETURN statement)

Usage

If the user highlights the menu item without choosing it, its Selected event occurs.

If the user chooses a menu item that has a cascaded menu associated with it, the Clicked event occurs, and the cascaded menu is displayed.
Examples

This script is for the Clicked event of the New menu item for the frame window. The `wf_newsheet` function is a window function. The window `w_genapp_frame` is part of the application template you can generate when you create a new application:

```powerscript
/* Create a new sheet */
w_genapp_frame.wf_newsheet();
```

See also

Selected

Syntax 2

For ListView controls

Description

Occurs when the user clicks within the ListView control, either on an item or in the blank space around items.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnclicked</td>
<td>ListView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value (the index of the ListView item the user clicked). The value of <code>index</code> is -1 if the user clicks within the control but not on a specific item.</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

0 Continue processing

Usage

The Clicked event occurs when the user presses the mouse button. The Clicked event can occur during a double-click, in addition to the DoubleClicked event.

In addition to the Clicked event, ItemChanging and ItemChanged events can occur when the user clicks on an item that does not already have focus. BeginLabelEdit can occur when the user clicks on a label of an item that has focus.

Using the ItemActivate event for ListView controls

You can use the ItemActivate event (with the OneClickActivate property set to true) instead of the Clicked event for ListView controls.
**Clicked**

**Examples**

This code changes the label of the item the user clicks to uppercase:

```powershell
IF index = -1 THEN RETURN 0

This.GetItem(index, llvi_current)
llvi_current.Label = Upper(llvi_current.Label)
This.SetItem(index, llvi_current)
RETURN 0
```

**See also**

ColumnClick
DoubleClicked
ItemActivate
ItemChanged
ItemChanging
RightClicked
RightDoubleClicked

**Syntax 3**

**For Tab controls**

**Description**

Occurs when the user clicks on the tab portion of a Tab control.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tcnclicked</td>
<td>Tab</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value (the index of the tab page the user clicked)</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0 Continue processing

**Usage**

The Clicked event occurs when the mouse button is released.

When the user clicks in the display area of the Tab control, the tab page user object (not the Tab control) gets a Clicked event.

The Clicked event can occur during a double-click, in addition to the DoubleClicked event.

In addition to the Clicked event, the SelectionChanging and SelectionChanged events can occur when the user clicks on a tab page label. If the user presses an arrow key to change tab pages, the Key event occurs instead of Clicked before SelectionChanging and SelectionChanged.
Examples

This code makes the tab label bold for the fourth tab page only:

```powerScript
IF index = 4 THEN
    This.BoldSelectedText = TRUE
ELSE
    This.BoldSelectedText = FALSE
END IF
```

See also

DoubleClicked
RightClicked
RightDoubleClicked
SelectionChanged
SelectionChanging

Syntax 4

For TreeView controls

Description

Occurs when the user clicks an item in a TreeView control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnclicked</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by value (the handle of the TreeView item the user clicked)</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

Usage

The Clicked event occurs when the user presses the mouse button.

The Clicked event can occur during a double-click, in addition to the DoubleClicked event.

In addition to the Clicked event, GetFocus occurs if the control does not already have focus.

Examples

This code in the Clicked event changes the label of the item the user clicked to uppercase:

```powerScript
TreeViewItem ltvi_current

This.GetItem(handle, ltvi_current)
ltvi_current.Label = Upper(ltvi_current.Label)
This.SetItem(handle, ltvi_current)
```
**Syntax 5**

**For windows and progress bars**

**Description**

Occurs when the user clicks in an unoccupied area of the window or progress bar (any area with no visible, enabled object).

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lbuttonclk</td>
<td>Window</td>
</tr>
<tr>
<td>pbm_lbuttondwn</td>
<td>HProgressBar, VProgressBar</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| flags    | UnsingedLong by value (the modifier keys and mouse buttons that are pressed). Values are:  
  • 1 – Left mouse button  
  • 2 – Right mouse button (windows only)  
  • 4 – Shift key  
  • 8 – Ctrl key  
  • 16 – Middle mouse button (windows only)  
  In the Clicked event for windows, the left mouse button is being released, so 1 is not summed in the value of flags. For an explanation of flags, see Syntax 2 of MouseMove on page 269. |
| xpos     | Integer by value (the distance of the pointer from the left edge of the window workspace or control in pixels). |
| ypos     | Integer by value (the distance of the pointer from the top of the window’s workspace or control in pixels). |

**Return codes**

Long. Return code choices (specify in a RETURN statement):

0  Continue processing
Usage

The Clicked event occurs when the user presses the mouse button down in progress bars and when the user releases the mouse button in windows.

If the user clicks on a control or menu in a window, that object (rather than the window) gets a Clicked event. No Clicked event occurs when the user clicks the window’s title bar.

When the user clicks on a window, the window’s MouseDown and MouseUp events also occur.

When the user clicks on a visible disabled control or an invisible enabled control, the window gets a Clicked event.

Examples

If the user clicks in the upper left corner of the window, this code sets focus to the button `cb_clear`:

```powerscript
IF (xpos <= 600 AND ypos <= 600) THEN
    cb_clear.SetFocus()
END IF
```

See also

DoubleClicked
MouseDown
MouseMove
MouseUp
RButtonDown

Syntax 6 For other controls

Description

Occurs when the user clicks on the control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_bnclicked</td>
<td>CheckBox, CommandButton, Graph, OLE, Picture, PictureHyperLink, PictureButton, RadioButton, StaticText, StaticHyperLink</td>
</tr>
<tr>
<td>pbm_lbuttondown</td>
<td>DatePicker, MonthCalendar</td>
</tr>
</tbody>
</table>

Arguments

None

Return codes

Long. Return code choices (specify in a RETURN statement):

- 0 Continue processing

Usage

The Clicked event occurs when the user releases the mouse button.

If another control had focus, then a GetFocus and a Clicked event occur for the control the user clicks.
Examples
This code in an OLE control’s Clicked event activates the object in the control:

```
integer li_success
li_success = This.Activate(InPlace!)
```

See also
GetFocus
RButtonDown

Close

The Close event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>OLE control</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Window</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

Syntax 1

For the application object

Description
Occurs when the user closes the application.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Application</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
None (do not use a RETURN statement)

Usage
The Close event occurs when the last window (for MDI applications the MDI frame) is closed.

See also
Open
SystemError
### Syntax 2  
**For OLE controls**

**Description**
Occurs when the object in an OLE control has been activated offsite (the OLE server displays the object in the server’s window) and that server is closed.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_omnclose</td>
<td>OLE</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code: Ignored

**Usage**
If the user closed the OLE server, the user’s choices might cause the OLE object in the control to be updated, triggering the Save or DataChange events.

If you want to retrieve the ObjectData blob value of an OLE control during the processing of this event, you must post a user event back to the control or you will generate a runtime error.

**See also**
DataChange
Save

### Syntax 3  
**For windows**

**Description**
Occurs just before a window is removed from display.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_close</td>
<td>Window</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code choices (specify in a RETURN statement):

0  Continue processing

**Usage**
When you call the Close function for the window, a CloseQuery event occurs before the Close event. In the CloseQuery event, you can specify a return code to prevent the Close event from occurring and the window from closing.

Do not trigger the Close event to close a window; call the Close function instead. Triggering the event simply runs the script and does not close the window.

**See also**
CloseQuery
Open
CloseQuery

Description
Occurs when a window is closed, before the Close event.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_closequery</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):

- 0  Allow the window to be closed
- 1  Prevent the window from closing

Usage
If the CloseQuery event returns a value of 1, the closing of the window is aborted and the Close event that usually follows CloseQuery does not occur.

If the user closes the window with the Close box (instead of using buttons whose scripts can evaluate the state of the data in the window), the CloseQuery event still occurs, allowing you to prompt the user about saving changes or to check whether data the user entered is valid.

Obsolete techniques
You no longer need to set the ReturnValue property of the Message object. Use a RETURN statement instead.

Examples
This statement in the CloseQuery event for a window asks if the user really wants to close the window and if the user answers no, prevents it from closing:

```powerbuilder
IF MessageBox("Closing window", "Are you sure?", &Question!, YesNo!) = 2 THEN
    RETURN 1
ELSE
    RETURN 0
END IF
```

This script for the CloseQuery event tests to see if the DataWindow dw_1 has any pending changes. If it has, it asks the user whether to update the data and close the window, close the window without updating, or leave the window open without updating:

```powerbuilder
integer li_rc

// Accept the last data entered into the datawindow
dw_1.AcceptText()
```
//Check to see if any data has changed
IF dw_1.DeletedCount()+dw_1.ModifiedCount() > 0 THEN
li_rc = MessageBox("Closing", &
"Update your changes?", Question!, &
YesNoCancel!, 3)

//User chose to up data and close window
IF li_rc = 1 THEN
Window lw_window
lw_window = w_genapp_frame.GetActiveSheet()
lw_window.TriggerEvent("ue_update")
RETURN 0

//User chose to close window without updating
ELSEIF li_rc = 2 THEN
RETURN 0

//User canceled
ELSE
RETURN 1
END IF

ELSE
// No changes to the data, window will just close
RETURN 0
END IF

See also Close

CloseUp

Description Occurs when the user has selected a date from the drop-down calendar and the calendar closes.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_dtpcloseup</td>
<td>DatePicker</td>
</tr>
</tbody>
</table>

Arguments None.

Return codes Long. Return code: Ignored.
**ColumnClick**

**Description**
Occurs when the user clicks a column header.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvncolumnclick</td>
<td>ListView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>column</td>
<td>The index of the clicked column</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specify in a RETURN statement):

0  Continue processing

**Usage**
The ColumnClicked event is only available when the ListView displays in report view and the ButtonHeader property is set to true.

**Examples**
This example uses the ColumnClicked event to set up a instance variable for the column argument, retrieve column alignment information, and display it to the user:

```powershell
string ls_label, ls_align
integer li_width
alignment la_align

ii_col = column
This.GetColumn(column, ls_label, la_align, &
li_width)

CHOOSE CASE la_align
CASE Right!
   rb_right.Checked = TRUE
   ls_align = "Right!"
CASE Left!
   rb_left.Checked = TRUE
   ls_align = "Left!"
CASE Center!
   rb_center.Checked = TRUE
   ls_align = "Center!"
CASE Justify!
   rb_just.Checked = TRUE
   ls_align = "Justify!"
END CHOOSE
```
ConnectMobiLink

Description
Occurs when the MobiLink synchronization server connects to the consolidated database server.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
None

Usage
When an application forms or reforms a connection with the MobiLink synchronization server, the MobiLink synchronization server temporarily allocates one connection with the database server for the duration of that synchronization.

Use the ConnectMobiLink event to add custom actions immediately before the remote database connects to the MobiLink synchronization server. At this stage, dbmlsync has generated the upload stream.

The following synchronization object events correspond to events occurring on the synchronization server (in the order displayed): BeginSync, ConnectMobiLink, BeginUpload, EndUpload, BeginDownload, EndDownload, DisconnectMobiLink, and EndSync.

See also
BeginDownload
BeginSync
BeginUpload
DisconnectMobiLink
Constructor

Description
Occurs when the control or object is created, just before the Open event for the window that contains the control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_constructor</td>
<td>All objects</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):

0 Continue processing

Usage
You can write a script for a control’s Constructor event to affect the control’s properties before the window is displayed.

When a window or user object opens, a Constructor event for each control in the window or user object occurs. The order of controls in a window’s Control property (which is an array) determines the order in which Constructor events are triggered. If one of the controls in the window is a user object, the Constructor events of all the controls in the user object occur before the Constructor event for the next control in the window.

When you call OpenUserObject to add a user object to a window dynamically, its Constructor event and the Constructor events for all of its controls occur.

When you use the CREATE statement to instantiate a class (nonvisual) user object, its Constructor event occurs.

When a class user object variable has an Autoinstantiate setting of true, its Constructor event occurs when the variable comes into scope. Therefore, the Constructor event occurs for:

- Global variables when the system starts up
- Shared variables when the object with the shared variables is loaded
- Instance variables when the object with the instance variables is created
- Local variables when the function that declares them begins executing

Examples
This example retrieves data for the DataWindow dw_1 before its window is displayed:

```pseudocode
    dw_1.SetTransObject(SQLCA)
    dw_1.Retrieve()
```

See also
Destructor
Open

208

PowerBuilder
**DataChange**

**Description**
Occurs when the server application notifies the control that data has changed.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_omndatachange</td>
<td>OLE</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code: Ignored

**See also**
PropertyChanged
Rename
ViewChange

---

**DateChanged**

**Description**
Occurs immediately after a date is selected.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_mcdatechanged</td>
<td>MonthCalendar</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code: Ignored

**Usage**
If you code a call to a MessageBox function in this event, the message box does not display if the user selects a new date using the mouse. This is because the mouse click captures the mouse. Message boxes do not display when the mouse is captured because unexpected results can occur. The message box does display if the user selects a new date using the arrow keys.

SetSelectedDate and SetSelectedRange trigger a DateChanged event. You should not call either method in a DateChanged event, particularly using the Post method.

**See also**
DateSelected
**DateSelected**

Description: Occurs when the user selects a date using the mouse.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_mcdatesel</td>
<td>MonthCalendar</td>
</tr>
</tbody>
</table>

Arguments: None

Return codes: Long. Return code: Ignored

Usage: This event is similar to DateChanged, but it occurs only when the user has selected a specific date using the mouse. The DateChanged event occurs whenever the date changes—when a date is selected using the mouse, when the date is changed in a script, and when the user uses the arrow key on the keyboard to select a different date or the arrow on the control to scroll to a different month.

Examples: The following script in the DateSelected event writes the date the user selected using the mouse to a single-line edit box:

```pseudocode
    date dt_selected
    integer li_ret
    string ls_date

    li_ret = GetSelectedDate( dt_selected)
    ls_date = string(dt_selected)
    s1e_2.text = ls_date
```

See also: DateChanged

**DBError**

Description: Triggered when an error occurs during a transaction or an attempted transaction.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Transaction objects</td>
</tr>
</tbody>
</table>
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>Long by value. A database-specific error code. See your DBMS documentation for information on the meaning of the code. When there is no error code from the DBMS, code contains one of these values: -1 – Cannot connect to the database -2 – Writing a blob to the database failed -4 – All other errors (see Usage note for more detail)</td>
</tr>
<tr>
<td>sqlerrortext</td>
<td>String by value. A database-specific error message.</td>
</tr>
<tr>
<td>sqlsyntax</td>
<td>String by value. The full text of the SQL statement being sent to the DBMS when the error occurred.</td>
</tr>
</tbody>
</table>

Return codes

Long, but this return code has no meaning to PowerBuilder.

Usage

Error codes For any database related error, the error code comes from the database driver. The error text is also from the database drivers. The sqlsyntax argument shows what SQL syntax was executing when the error occurred.

For errors that are not related to database drivers, the code argument is set to -4. If the PowerBuilder VM cannot get the syntax for these types of errors, an empty string is passed to the sqlsyntax argument. PowerBuilder cannot get the syntax for the following types of errors:

- “Cursor is not open”
- “Procedure has not been executed or has no results”
- “Transaction not connected”
- “Transaction not connected. Transaction Pool limit exceeded”
- “Transaction already connected”
- “Database does not support FETCH (FIRST/LAST/PRIOR)”
- “Select returned more than one row”
- “Mismatch between prepared number of substitution variables and execute parameters”
- “Blob variable for UPDATEBLOB cannot be empty”
- “Open <cursor> or execute <procedure> must reference DESCRIPTOR”

The PowerBuilder VM can get the SQL syntax for the following types of errors, and passes it to the Transaction object’s DBError event for the following types of errors:
Use with embedded SQL  By default, whenever an error occurs in the Transaction object, the DBError event is called. The error code and error message are passed to this event. You can add code to the DBError event to handle these errors.

Use with DataWindow/DataStore  When using a Transaction object with a DataWindow, the DataWindow DBError event is triggered before the DBError event of the Transaction object. The return value for the DataWindow DBError event is used to indicate whether the Transaction object’s DBError event should be triggered in turn. When the return value of the DataWindow DBError event is 0 or 1, the Transaction object’s DBError event is also triggered if it is defined. When the return value of the DataWindow DBError event is 2 or 3, the Transaction object’s DBError event is ignored.

Examples  The following code in the DBError event displays the error message and the SQL statement sent to the DBMS when a transaction error occurs:

```
Messagebox("Transaction error","Error message: "+sqlerrortext +"~r~n Occurred for this statement:"+sqlsyntax)
```

See also  DBError in the DataWindow Reference

SQLPreview

### DBNotification

**Description**

Triggered by a PowerBuilder script or DataWindow database operation command if a PowerBuilder database driver receives a notification from the database server. This event is supported only with the Oracle 10g (O10) native database interface.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_dbnotication</td>
<td>Transaction</td>
</tr>
</tbody>
</table>
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| notification| A value of the DBNotification enumerated datatype. The database interface determines the type of the notification received from the server, triggers the DBNotification event, and passes the notification type in this argument. Values are:
  • DBServerDown! = 1. The server has been shut down. This notification type is used only by the O10 (Oracle 10g) database interface.
  • DBFailover! = 2. The database client is failing over.
  • DBDataTruncate! = 3. Data has been truncated. DBServerDown! is used for Oracle RAC database HA events. DBFailover! and DBDataTruncate! can be used with other databases for failover and data truncation warnings. |

| command     | A string that informs users which command was being executed when the notification occurred.                                                 |
| dbmessage   | A string that describes the reason why the event occurred.                                                                                   |

Return codes

Long. Return code choices (specify in a RETURN statement):

• 0 Continue to process the database command. If the event does not exist or does not have a script, the return value is 0 by default.

• Any other value Ignored if the notification argument is DBFailover!. If the value of the notification argument is DBServerDown! or DBDataTruncate!, the current command returns with an error. SQLCA.SQLCode is set to -1 and SQLCA.SQLDBCCode is set to the return value.

Usage

Oracle Real Application Clusters (RAC) is a cluster database that uses a shared cache architecture. In Oracle 10g Release 2, a High Availability (HA) client connected to an RAC database can register a callback to indicate that it wants the server to notify it in case of a database failure event that affects a connection made by the client. The DBNotification event is triggered when the client is notified that such an event has occurred.

The default transaction object, SQLCA, does not support this event. To use the event, create a new standard class user object that inherits from the Transaction object and add code to the DBNotification event script. You can then use this Transaction object in your application, or substitute it for SQLCA on the Variable Types tab page in the Application Properties dialog box.
To be notified when the server shuts down, your application must be connected to an Oracle 10g RAC database using the O10 database interface and the HANotification database parameter must be set to 1. When the server shuts down, the O10 driver is notified. The DBNotification event is triggered if the application continues to attempt to access the server. The value of the notification argument is set to DBServerDown!, the command string is set to the syntax of the current command, and the dbmessage string is populated with information about the shutdown.

If your application does not execute any SQL statements on the current connection after the server shuts down, the DBNotification event is not triggered until Disconnect is called.

You can code the return value of the DBNotification event to specify whether the application should continue to execute the current command:

- If the event returns 0, the current command continues executing until failover occurs and completes successfully (if failover is supported), then the application continues. If failover is not supported, the application will receive an error for the current command.

- If the event returns any other value, the current command execution is stopped immediately and the Transaction object property SQLCode is set to -1, SQLDBCode is set to the return value, SQLErrText is set to the value of the dbmessage string, and failover does not happen. After the event, only Disconnect can be called on the current transaction.

Inside the DBNotification event script, the current connection of the Transaction object is protected and database operations with the connection are not allowed. All database commands will return as failed. However, the application can still access the database with another Transaction object.

If the SvrFailover database parameter is set to Yes, the DBNotification event is triggered with the notification argument set to DBFailover!. The event can be triggered several times during the failover, for example when the failover begins and ends. You do not need to be connected to an Oracle RAC database or to set the HANotification database parameter to be notified when a failover occurs.
**Deactivate**

Description: Occurs when the window becomes inactive.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_deactivate</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments: None

Return codes: Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>

Usage: When a window is closed, a Deactivate event occurs.

See also: Activate, Show

---

**DeleteAllItems**

Description: Occurs when all the items in the ListView are deleted.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvndeleteallitems</td>
<td>ListView</td>
</tr>
</tbody>
</table>

Arguments: None

Return codes: Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>

Examples: This example uses the DeleteAllItems event to ensure that there is a default item in the ListView control:

```
This.AddItem("Default item", 1)
```

See also: DeleteItem, InsertItem
DeleteItem

The DeleteItem event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For ListView controls**

Description

Occurs when an item is deleted.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvndeleteitem</td>
<td>ListView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value (the index of the deleted item)</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

Examples

This example for the DeleteItem event displays a message with the number of the deleted item:

```powershell
MessageBox("Message", "Item " + String(index) & 
            + " deleted.")
```

See also

DeleteAllItems
InsertItem

**Syntax 2**

**For TreeView controls**

Description

Occurs when an item is deleted.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvndeleteitem</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by value (the handle of the deleted item)</td>
</tr>
</tbody>
</table>
Return codes Long. Return code choices (specify in a RETURN statement):
   0 Continue processing

Examples This example displays the name of the deleted item in a message:
   TreeViewItem ll_tvi
   This.GetItem(handle, ll_tvi)
   MessageBox("Message", String(ll_tvi.Label) + " has been deleted.")

**Destructor**

Description Occurs when the user object or control is destroyed, immediately after the Close event of a window.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_destructor</td>
<td>All objects</td>
</tr>
</tbody>
</table>

Arguments None

Return codes Long. Return code choices (specify in a RETURN statement):
   0 Continue processing

Usage When a window is closed, each control’s Destructor event destroys the control and removes it from memory. After they have been destroyed, you can no longer refer to those controls in other scripts. If you do, a runtime error occurs.

See also Constructor Close

**DisconnectMobiLink**

Description Occurs when the MobiLink synchronization server disconnects from the consolidated database server.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>
DisplayMessage

Arguments
None

Return codes
None

Usage
Use this event to add custom actions immediately after the remote database disconnects from the MobiLink synchronization server.

When an application forms or reforms a connection with the MobiLink synchronization server, the MobiLink synchronization server temporarily allocates one connection with the database server for the duration of that synchronization.

The following synchronization object events correspond to events occurring on the synchronization server (in the order displayed): BeginSync, ConnectMobiLink, BeginUpload, EndUpload, BeginDownload, DisconnectMobiLink, and EndSync.

See also
ConnectMobiLink
EndDownload
EndSync
EndUpload

DisplayMessage

Description
Occurs on display of an informational message from a MobiLink synchronization.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>infomsg</td>
<td>Read-only string containing the text of an informational message returned from the synchronization server.</td>
</tr>
</tbody>
</table>

Return codes
None

Usage
The following events are triggered when different types of messages are sent by the synchronization server: DisplayMessage, ErrorMessage, FileMessage, and WarningMessage.

See also
ErrorMessage
FileMessage
WarningMessage
DoubleClicked

The DoubleClicked event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListBox, PictureListBox, ListView, and Tab controls</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Window</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>Other controls</td>
<td>Syntax 4</td>
</tr>
</tbody>
</table>

For information about the DataWindow control’s DoubleClicked event, see the DataWindow Reference or the online Help.

Syntax 1

For ListBox, PictureListBox, ListView, and Tab controls

Description

Occurs when the user double-clicks on the control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lbndbleclk</td>
<td>ListBox, PictureListBox</td>
</tr>
<tr>
<td>pbm_lvndoubleclicked</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_tcndoubleclicked</td>
<td>Tab</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value. The index of the item the user double-clicked (for tabs, the index of the tab page).</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage

You can use the ItemActivate event (with the OneClickActivate property set to false) instead of the DoubleClicked event for ListView controls.

In a ListBox or PictureListBox, double-clicking on an item also triggers a SelectionChanged event.

Examples

This example uses the DoubleClicked event to begin editing the double-clicked ListView item:

This.EditLabels = TRUE
DoubleClicked

See also
Clicked
ColumnClick
ItemActivate
ItemChanged
ItemChanging
RightClicked
RightDoubleClicked
SelectionChanged
SelectionChanging

Syntax 2 For TreeView controls
Description
Occurs when the user double-clicks on the control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvndoubleclicked</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by value (the handle of the item the user double-clicked)</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):

0    Continue processing

Examples
This example turns on editing for the double-clicked TreeView item:

```
TreeViewItem ltvi_current
ltvi_current = tv_1.FindItem(CurrentTreeItem!, 0)
This.EditLabel(ltvi_current)
```

See also
Clicked
RightClicked
RightDoubleClicked
SelectionChanged
SelectionChanging
### Syntax 3

#### For windows

**Description**

Occurs when the user double-clicks in an unoccupied area of the window (any area with no visible, enabled object).

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lbuttondblck</td>
<td>Window</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| flags    | UnsingedLong by value (the modifier keys and mouse buttons that are pressed). Values are:  
  • 1 – Left mouse button  
  • 2 – Right mouse button  
  • 4 – Shift key  
  • 8 – Ctrl key  
  • 16 – Middle mouse button  
  In the Clicked event, the left mouse button is being released, so 1 is not summed in the value of flags.  
  For an explanation of flags, see Syntax 2 of MouseMove on page 269. |
| xpos     | Integer by value (the distance of the pointer from the left edge of the window’s workspace in pixels). |
| ypos     | Integer by value (the distance of the pointer from the top of the window’s workspace in pixels). |

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0 Continue processing

**Usage**

The xpos and ypos arguments provide the same values the functions PointerX and PointerY return when you call them for the window.

**See also**

Clicked  
MouseDown  
MouseMove  
MouseUp  
RButtonDown
### DragDrop

**Syntax 4**

**For other controls**

Occurs when the user double-clicks on the control.

#### Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_bndoubleclicked</td>
<td>Graph, OLE, Picture, PictureHyperLink, StaticText, StaticHyperLink</td>
</tr>
<tr>
<td>pbm_cbndblclk</td>
<td>DropDownListBox, DropDownListBox, DropDownListBox</td>
</tr>
<tr>
<td>pbm_lbuttondblclk</td>
<td>DatePicker, MonthCalendar</td>
</tr>
<tr>
<td>pbm_prndoubleclicked</td>
<td>HProgressBar, VProgressBar</td>
</tr>
<tr>
<td>pbm_rendoubleclicked</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

**Arguments**

None

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0 Continue processing

**Usage**

The DoubleClicked event for DropDownListBoxes is only active when the Always Show List property is on.

**See also**

Clicked

RButtonDown

---

### DragDrop

The DragDrop event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListBox, PictureListBox, ListView, and Tab controls</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Windows and other controls</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

For information about the DataWindow control’s DragDrop event, see the *DataWindow Reference or the online Help.*
### Syntax 1

**For ListBox, PictureListBox, ListView, and Tab controls**

**Description**

Occurs when the user drags an object onto the control and releases the mouse button to drop the object.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lbndragdrop</td>
<td>ListBox, PictureListBox</td>
</tr>
<tr>
<td>pbm_lvndragdrop</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_tcndragdrop</td>
<td>Tab</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>DragObject by value (a reference to the control being dragged)</td>
</tr>
<tr>
<td>index</td>
<td>Integer by value (the index of the target ListView item)</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0 Continue processing

**Usage**

*Obsolete functions* You no longer need to call the **DraggedObject** function in a drag event. Use the `source` argument instead.

**Examples**

For ListView controls, see the example for **BeginDrag**.

This example inserts the dragged ListView item:

```powerscript
This.AddItem(ilvi_dragged_object)
This.Arrange();
```

**See also**

BeginDrag
BeginRightDrag
DragEnter
DragLeave
DragWithin
### Syntax 2

**For TreeView controls**

**Description**

Occurs when the user drags an object onto the control and releases the mouse button to drop the object.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvndragdrop</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>DragObject by value (a reference to the control being dragged)</td>
</tr>
<tr>
<td>handle</td>
<td>Long by value (the handle of the target TreeView item)</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0 Continue processing

**Usage**

*Obsolete functions* You no longer need to call the DraggedObject function in a drag event. Use the *source* argument instead.

**Examples**

This example inserts the dragged object as a child of the TreeView item it is dropped upon:

```powerbuilder
TreeViewItem ltv_1
This.GetItem(handle, ltv_1)
This.SetDropHighlight(handle)
This.InsertItemFirst(handle, itvi_drag_object)
This.ExpandItem(handle)
This.SetRedraw(TRUE)
```

**See also**

DragEnter
DragLeave
DragLeave

### Syntax 3

**For windows and other controls**

**Description**

Occurs when the user drags an object onto the control and releases the mouse button to drop the object.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_bndragdrop</td>
<td>CheckBox, CommandButton, Graph, InkEdit, InkPicture, Picture, PictureHyperLink, PictureButton, RadioButton</td>
</tr>
<tr>
<td>pbm_cbndragdrop</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
</tbody>
</table>
CHAPTER 9  PowerScript Events

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>DragObject by value (a reference to the control being dragged)</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage

When a control’s DragAuto property is true, a drag operation begins when the user presses a mouse button.

Obsolete functions  You no longer need to call the DraggedObject function in a drag event. Use the source argument instead.

Examples

Example 1  In this example, the code in the DoubleClicked event for the DataWindow dw_orddetail starts a drag operation:

```
IF dw_orddetail.GetRow() > 0 THEN
    dw_orddetail.Drag(Begin!)
    This.DragIcon = "dragitem.ico"
END IF
```

Then, in the DragDrop event for a trashcan Picture control, this code deletes the row the user clicked and dragged from the DataWindow control:

```
long ll_currow
dwitemstatus ldwis_delrow

ll_currow = dw_orddetail.GetRow()
// Save the row's status flag for later use
ldwis_delrow = dw_orddetail.GetItemStatus & (ll_currow, 0, Primary!)
```
// Now, delete the current row from dw_orddetail
dw_orddetail.DeleteRow(0)

**Example 2**  This example for a trashcan Picture control’s DragDrop event checks whether the source of the drag operation is a DataWindow. If so, it asks the user whether to delete the current row in the source DataWindow:

```powerbuilder
DataWindow ldw_Source
Long li_RowToDelete
Integer li_Choice

IF source.TypeOf() = DataWindow! THEN

    ldw_Source = source
    li_RowToDelete = ldw_Source.GetRow()

    IF li_RowToDelete > 0 THEN
        li_Choice = MessageBox("Delete", & 
            "Delete this row?", Question!, YesNo!, 2)

        IF li_Choice = 1 THEN
            ldw_Source.DeleteRow(li_RowToDelete)
            END IF

        ELSE
            Beep(1)
            END IF

    ELSE
        Beep(1)
        END IF

ELSE
    Beep(1)
END IF
```

See also DragEnter
DragLeave
DragWithin
CHAPTER 9  PowerScript Events

DragEnter

Description

Occurs when the user is dragging an object and enters the control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_bndragenter</td>
<td>CheckBox, CommandButton, Graph, InkEdit, InkPicture, Picture, PictureHyperlink, PictureButton, RadioButton</td>
</tr>
<tr>
<td>pbm_cbndragenter</td>
<td>DropDownListBox, DropDownListBox</td>
</tr>
<tr>
<td>pbm_dragenter</td>
<td>DatePicker, MonthCalendar</td>
</tr>
<tr>
<td>pbm_dndragenter</td>
<td>DataWindow</td>
</tr>
<tr>
<td>pbm_endragenter</td>
<td>SingleLineEdit, EditMask, MultiLineEdit, StaticText, StaticHyperLink</td>
</tr>
<tr>
<td>pbm_lbdragenter</td>
<td>ListBox, PictureListBox</td>
</tr>
<tr>
<td>pbm_lvndragenter</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_omndragenter</td>
<td>OLE</td>
</tr>
<tr>
<td>pbm_prndragenter</td>
<td>HProgressBar, VProgressBar</td>
</tr>
<tr>
<td>pbm_rndragenter</td>
<td>RichTextEdit</td>
</tr>
<tr>
<td>pbm_sbdragenter</td>
<td>HScrollBar, HTrackBar, VScrollBar, VTrackBar</td>
</tr>
<tr>
<td>pbm_tcdragenter</td>
<td>Tab</td>
</tr>
<tr>
<td>pbm_tvndragenter</td>
<td>TreeView</td>
</tr>
<tr>
<td>pbm_uondragenter</td>
<td>UserObject</td>
</tr>
<tr>
<td>pbm_dragenter</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>DragObject by value (a reference to the control being dragged)</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage

Obsolete functions  You no longer need to call the DraggedObject function in a drag event. Use the source argument instead.

Examples

This example for a Picture control’s DragDrop event adds a border to itself when another Picture control (the source) is dragged within its boundaries:

```powerScript
IF source.TypeOf() = Picture! THEN
  This.Border = TRUE
END IF
```

See also

DragDrop
DragLeave
DragWithin

PowerScript Reference 227
DragLeave

Description
Occurs when the user is dragging an object and leaves the control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_bndragleave</td>
<td>CheckBox, CommandButton, Graph, InkEdit, InkPicture,</td>
</tr>
<tr>
<td></td>
<td>Picture, PictureHyperLink, PictureButton, RadioButton</td>
</tr>
<tr>
<td>pbm_cbndragleave</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_dragleave</td>
<td>DatePicker, MonthCalendar</td>
</tr>
<tr>
<td>pbm_dwndragleave</td>
<td>DataWindow</td>
</tr>
<tr>
<td>pbm_endragleave</td>
<td>SingleLineEdit, EditText, MultiLineEdit, StaticText,</td>
</tr>
<tr>
<td></td>
<td>StaticHyperLink</td>
</tr>
<tr>
<td>pbm_lbdragleave</td>
<td>ListBox, PictureListBox</td>
</tr>
<tr>
<td>pbm_lvndragleave</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_omndragleave</td>
<td>OLE</td>
</tr>
<tr>
<td>pbm_prndragleave</td>
<td>HProgressBar, VProgressBar</td>
</tr>
<tr>
<td>pbm_rndragleave</td>
<td>RichTextEdit</td>
</tr>
<tr>
<td>pbm_sbdragleave</td>
<td>HScrollBar, HTrackBar, VScrollBar, VScrollBar</td>
</tr>
<tr>
<td>pbm_tcndragleave</td>
<td>Tab</td>
</tr>
<tr>
<td>pbm_tvndragleave</td>
<td>TreeView</td>
</tr>
<tr>
<td>pbm_uondragleave</td>
<td>UserObject</td>
</tr>
<tr>
<td>pbm_dragleave</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>DragObject by value (a reference to the control being dragged)</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):

0    Continue processing

Usage
Obsolete functions    You no longer need to call the DraggedObject function in a drag event. Use the source argument instead.

Examples
This example checks the name of the control being dragged, and if it is, cb_1 it cancels the drag operation:

```
IF ClassName(source) = "cb_1" THEN
  cb_1.Drag(Cancel!)
END If
```
This example for a Picture control’s DragDrop event removes its own border when another Picture control (the source) is dragged beyond its boundaries:

\[
\text{IF source.TypeOf()} = \text{Picture}! \amp \text{THEN} \\
\quad \quad \text{This.Border} = \text{TRUE} \\
\text{END IF}
\]

See also

DragDrop
DragEnter
DragWithin

**DragWithin**

The DragWithin event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListBox, PictureListBox, ListView, and Tab controls</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Windows and other controls</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

For information about the DataWindow control’s DragWithin event, see the DataWindow Reference or the online Help.

**Syntax 1**

**For ListBox, PictureListBox, ListView, and Tab controls**

Occurs when the user is dragging an object within the control.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lbndragwithin</td>
<td>ListBox, PictureListBox</td>
</tr>
<tr>
<td>pbm_lvndragwithin</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_tcdragwithin</td>
<td>Tab</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>DragObject by value (a reference to the control being dragged)</td>
</tr>
<tr>
<td>index</td>
<td>Integer by value (a reference to the ListView item under the pointer in the ListView control)</td>
</tr>
</tbody>
</table>
**DragWithin**

Return codes

Long. Return code choices (specify in a RETURN statement):

0 Continue processing

Usage

*Obsolete functions* You no longer need to call the DraggedObject function in a drag event. Use the *source* argument instead.

Examples

This example changes the background color of the ListView when a DragObject enters its border:

```
This.BackColor = RGB(128, 0, 128)
```

See also

DragDrop
DragEnter
DragLeave

**Syntax 2**

For TreeView controls

Description

Occurs when the user is dragging an object within the control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvndragwithin</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>source</em></td>
<td>DragObject by value (a reference to the control being dragged)</td>
</tr>
<tr>
<td><em>handle</em></td>
<td>Long (a reference to the ListView item under the pointer in the TreeView control)</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

0 Continue processing

Usage

*Obsolete functions* You no longer need to call the DraggedObject function in a drag event. Use the *source* argument instead.

Examples

This example changes the background color of the TreeView when a DragObject enters its border:

```
This.BackColor = RGB(128, 0, 128)
```

See also

DragDrop
DragEnter
DragLeave
**Syntax 3**

**For windows and other controls**

Occurs when the user is dragging an object within the control.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_bndragwithin</td>
<td>CheckBox, CommandButton, Graph, InkEdit, InkPicture, Picture, PictureHyperLink, PictureButton, RadioButton</td>
</tr>
<tr>
<td>pbm_cbdragwithin</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_dragwithin</td>
<td>DatePicker, MonthCalendar</td>
</tr>
<tr>
<td>pbm_endragwithin</td>
<td>SingleLineEdit, EditMask, MultiLineEdit, StaticText, StaticHyperLink</td>
</tr>
<tr>
<td>pbm_omndragwithin</td>
<td>OLE</td>
</tr>
<tr>
<td>pbm_prndragwithin</td>
<td>HProgressBar, VProgressBar</td>
</tr>
<tr>
<td>pbm_rendragwithin</td>
<td>RichTextEdit</td>
</tr>
<tr>
<td>pbm_sbndragwithin</td>
<td>HScrollBar, HTrackBar, VScrollBar, VTrackBar</td>
</tr>
<tr>
<td>pbm_uondragwithin</td>
<td>UserObject</td>
</tr>
<tr>
<td>pbm_dragwithin</td>
<td>Window</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>DragObject by value (a reference to the control being dragged)</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Usage**

*Obsolete functions* You no longer need to call the DraggedObject function in a drag event. Use the *source* argument instead.

**See also**

DragDrop
DragEnter
DragLeave
DropDown

Description
Occurs when the user has clicked the drop-down arrow in a DatePicker control just before the drop-down calendar displays.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_dtpdropdown</td>
<td>DatePicker</td>
</tr>
</tbody>
</table>

Arguments
None.

Return codes
Long. Return code: Ignored.

EndDownload

Description
Occurs at the end of a download procedure

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upsertrows</td>
<td>Long identifying the inserted and updated rows.</td>
</tr>
<tr>
<td>deleterows</td>
<td>Long identifying the deleted rows.</td>
</tr>
</tbody>
</table>

Return codes
None

Usage
Use this event to add custom actions at the end of the download stage of synchronization.

The BeginDownload event marks the beginning of the download transaction. Applicable deletes are performed for all remote tables, and then rows are added as applicable for all remote tables in the download cursor. After EndDownload, download changes are committed.

See also
BeginDownload
ConnectMobiLink
EndSync
EndUpload
EndLabelEdit

The EndLabelEdit event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For ListView controls**

Occurs when the user finishes editing an item’s label.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnendlabeledit</td>
<td>ListView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer. The index of the ListView item for which you have edited the label.</td>
</tr>
<tr>
<td>newlabel</td>
<td>The string that represents the new label for the ListView item.</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

0  Allow the new text to become the item’s label.
1  Prevent the new text from becoming the item’s label.

**Usage**

The user triggers this event by pressing Enter or Tab after editing the text.

**Examples**

This example displays the old label and the new label in a SingleLineEdit:

```power_script
ListViewItem lvi
sle_info.text = "Finished editing " &
  + String(lvi.label) &
  + ". Item changed to "+ String(newlabel)
```

**See also**

BeginLabelEdit
Syntax 2 For TreeView controls

Description
Occurs when the user finishes editing an item’s label.

Event ID
- **Event ID**: pbm_tvnendlabeledit
- **Objects**: TreeView

Arguments
- **Argument**: handle
  - **Description**: Integer. The index of the TreeView item for which you have edited the label.
- **Argument**: newtext
  - **Description**: The string that represents the new label for the TreeView item.

Return codes
Long. Return code choices (specify in a RETURN statement):
- 0 Allow the new text to become the item’s label
- 1 Prevent the new text from becoming the item’s label

Usage
The user triggers this event by pressing Enter or Tab after editing the text.

Examples
This example displays the old label and the new label in a SingleLineEdit:

```
TreeViewItem tvi

This.GetItem(handle, tvi)
sle_info.Text = "Finished editing " &
+ String(tvi.Label) &
+ ", Item changed to " &
+ String(newtext)
```

See also
BeginLabelEdit

EndLogScan

Description
Occurs after the scan of the transaction log completes for upload.

Event ID
- **Event ID**: None
- **Objects**: MLSync

Arguments
None

Return codes
None
Usage

Use this event to add custom actions immediately after the transaction log is scanned for upload.

The following events are triggered while the upload stream is prepared, but before synchronization begins: BeginLogScan, ProgressInfo, and EndLogScan.

See also

BeginLogScan
ProgressIndex

EndSync

Description

Occurs at the end of synchronization.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>Long datatype value that indicates whether a synchronization error occurred.</td>
</tr>
<tr>
<td>restart</td>
<td>Boolean value passed by reference that, if true, causes dbmlsync to restart the synchronization.</td>
</tr>
</tbody>
</table>

Return codes

None

Usage

Use this event to add custom actions when a synchronization is complete.

An rc value of 0 indicates a successful synchronization. When the rc value is anything other than 0, an error has occurred. If the restart value changes to true, dbmlsync restarts the synchronization.

See also

BeginSync
DisconnectMobiLink
EndDownload
EndUpload
**EndUpload**

**Description**
Occurs after transmission of the upload to the synchronization server.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
None

**Usage**
Use this event to add custom actions immediately after transmission of the upload stream from dbmlsync to the MobiLink synchronization server.

The BeginUpload event marks the beginning of the upload transaction. Applicable inserts and updates to the consolidated database are performed for all remote tables, then rows are deleted as applicable for all remote tables. After EndUpload, upload changes are committed.

**See also**
BeginUpload
DisconnectMobiLink
EndDownload
EndSync

---

**Error**

**Description**
Occurs when an error is found in a data or property expression for an external object or a DataWindow object. Also occurs when a communications error is found in a client connecting to EAServer.

---

**Improved error-handling capability in PowerBuilder**
The Error event is maintained for backward compatibility. If you do not script the Error event or change its action argument, information from this event is passed to RuntimeException objects, such as DWRuntimeError or OLERuntimeError. You can handle these errors in a try-catch block.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Connection, DataWindow, DataStore, JaguarORB, OLE, OLEObject, OLETxnObject</td>
</tr>
</tbody>
</table>
### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>errornumber</code></td>
<td>Unsigned integer by value (PowerBuilder’s error number)</td>
</tr>
<tr>
<td><code>errortext</code></td>
<td>String, read-only (PowerBuilder’s error message)</td>
</tr>
<tr>
<td><code>errorwindowmenu</code></td>
<td>String, read-only (the name of the window or menu that is</td>
</tr>
<tr>
<td></td>
<td>the parent of the object whose script caused the error)</td>
</tr>
<tr>
<td><code>errorobject</code></td>
<td>String, read-only (the name of the object whose script caused the error)</td>
</tr>
<tr>
<td><code>errorscript</code></td>
<td>String, read-only (the full text of the script in which the</td>
</tr>
<tr>
<td></td>
<td>error occurred)</td>
</tr>
<tr>
<td><code>errorline</code></td>
<td>Unsigned integer by value (the line in the script where the</td>
</tr>
<tr>
<td></td>
<td>error occurred)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>action</code></th>
<th>ExceptionAction by reference.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>A value you specify to control the application’s course of action as a</td>
</tr>
<tr>
<td>Specification</td>
<td>result of the error. Values are:</td>
</tr>
<tr>
<td></td>
<td>• ExceptionFail! – fail as if this script were not implemented. The error</td>
</tr>
<tr>
<td></td>
<td>condition triggers any active event handlers, or if none, the SystemError</td>
</tr>
<tr>
<td></td>
<td>event.</td>
</tr>
<tr>
<td></td>
<td>• ExceptionIgnore! – ignore this error and return as if no error occurred</td>
</tr>
<tr>
<td></td>
<td>(use this option with caution because the conditions that caused the error</td>
</tr>
<tr>
<td></td>
<td>can cause another error).</td>
</tr>
<tr>
<td></td>
<td>• ExceptionRetry! – execute the function or evaluate the expression again</td>
</tr>
<tr>
<td></td>
<td>in case the OLE server was not ready. This option is not valid for</td>
</tr>
<tr>
<td></td>
<td>DataWindows.</td>
</tr>
<tr>
<td></td>
<td>• ExceptionSubstituteReturnValue! – use the value specified in the</td>
</tr>
<tr>
<td></td>
<td><code>returnvalue</code> argument instead of the value returned by the OLE server or</td>
</tr>
<tr>
<td></td>
<td>DataWindow, and cancel the error condition.</td>
</tr>
</tbody>
</table>

| `returnvalue`   | Any by reference (a value whose datatype matches the expected value that    |
|                 | the OLE server or DataWindow would have returned).                          |
|                 | This value is used when the value of `action` is                           |
|                 | ExceptionSubstituteReturnValue!.                                           |

### Return codes
None. Do not use a RETURN statement.

### Usage
DataWindow and OLE objects are dynamic. Expressions that use dot notation to refer to data and properties of these objects might be valid under some runtime conditions but not others. The Error event allows you to respond to this dynamic situation with error recovery logic.
The Error event also allows you to respond to communications errors in the client component of a distributed application. In the Error event for a custom connection object, you can tell PowerBuilder what action to take when an error occurs during communications between the client and the server.

The Error event gives you an opportunity to substitute a default value when the error is not critical to your application. Its arguments also provide information that is helpful in debugging. For example, the arguments can help you debug DataWindow data expressions that cannot be checked by the compiler—such expressions can only be evaluated at runtime.

**When to substitute a return value**

The ExceptionSubstituteReturnValue! action allows you to substitute a return value when the last element of an expression causes an error. Do not use it to substitute a return value when an element in the middle of an expression causes an error. The substituted return value does not match the datatype of the unresolved object reference and causes a system error.

The ExceptionSubstituteReturnValue! action can be useful for handling errors in data expressions.

For DataWindows, when an error occurs while evaluating a data or property expression, error processing occurs like this:

1. The Error event occurs.
2. If the Error event has no script or its action argument is set to ExceptionFail!, any active exception handler for a DWRuntimeError or its RuntimeError ancestor is invoked.
3. If no exception handler exists, or if the existing exception handlers do not handle the exception, the SystemError event is triggered.
4. If the SystemError event has no script, an application error occurs and the application is terminated.

The error processing in the client component of a distributed application is the same as for DataWindows.

For information about error processing in OLE controls, see the ExternalException event. For information about data and property expressions for DataWindow objects, see the DataWindow Reference or the online Help.

For information about handling communications errors in a multitier application, see the discussion of distributed applications in Application Techniques.
Examples

This example displays information about the error that occurred and allows the script to continue:

```
MessageBox("Error Number " + string(errornumber) &
  + " Occurred", "Errortext: " + String(errortext))
action = ExceptionIgnore!
```

See also

DBError in the DataWindow Reference or the online Help
ExternalException
SystemError

**ErrorMessage**

**Description**

Occurs on display of an error message from a MobiLink synchronization.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errmsg</td>
<td>Read-only string containing the text of the error message returned from the synchronization server.</td>
</tr>
</tbody>
</table>

**Return codes**

None

**Usage**

Use this event to receive error information logged by dbmlsync.

The following events can be triggered when different types of messages are sent by the synchronization server: DisplayMessage, ErrorMessage, FileMessage, and WarningMessage.

See also

DisplayMessage
FileMessage
WarningMessage
**ExternalException**

**Description**
Occurs when an OLE automation command caused an exception on the OLE server.

**Improved error-handling capability in PowerBuilder**
The ExternalException event is maintained for backward compatibility. If you do not script this event or change its action argument, information from this event is passed to RuntimeError objects, such as OLERuntimeError. You can handle these errors in a try-catch block.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>OLE, OLEObject, OLETxnObject</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resultcode</td>
<td>UnsignedLong by value (a PowerBuilder number identifying the exception that occurred on the server).</td>
</tr>
<tr>
<td>exceptioncode</td>
<td>UnsignedLong by value (a number identifying the error that occurred on the server. For the meaning of the code, see the server documentation).</td>
</tr>
<tr>
<td>source</td>
<td>String by value (the name of the server, which the server provides).</td>
</tr>
<tr>
<td>description</td>
<td>String by value (a description of the exception, which the server provides).</td>
</tr>
<tr>
<td>helpfile</td>
<td>String by value (the name of a Help file containing information about the exception, which the server provides).</td>
</tr>
<tr>
<td>helpcontext</td>
<td>UnsignedLong by value (the context ID of a Help topic in helpfile containing information about the exception, which the server provides).</td>
</tr>
</tbody>
</table>
### Return codes
None. (Do not use a RETURN statement.)

### Usage
OLE objects are dynamic. Expressions that refer to data and properties of these objects might be valid under some runtime conditions but not others. If the expression causes an exception on the server, PowerBuilder triggers the ExternalException event. The ExternalException event gives you information about the error that occurred on the OLE server.

The server defines what it considers exceptions. Some errors, such as mismatched datatypes, generally do not cause an exception but do trigger the Error event. In some cases you might not consider the cause of the exception to be an error. To determine the reason for the exception, see the documentation for the server.

When an exception occurs because of a call to an OLE server, error handling occurs like this:

1. The ExternalException event occurs.
2. If the ExternalException event has no script or its `action` argument is set to ExceptionFail!, the Error event occurs.
3. If the Error event has no script or its `action` argument is set to ExceptionFail!, any active exception handler for an OLERuntimeError or its RuntimeError ancestor is invoked.

### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>action</code></td>
<td>ExceptionAction by reference. A value you specify to control the application’s course of action as a result of the error. Values are:</td>
</tr>
<tr>
<td></td>
<td>- ExceptionFail! – fail as if this script were not implemented. The error condition triggers the SystemError event.</td>
</tr>
<tr>
<td></td>
<td>- ExceptionIgnore! – ignore this error and return as if no error occurred (use this option with caution because the conditions that caused the error can cause another error).</td>
</tr>
<tr>
<td></td>
<td>- ExceptionRetry! – execute the function or evaluate the expression again in case the OLE server was not ready.</td>
</tr>
<tr>
<td></td>
<td>- ExceptionSubstituteReturnValue! – use the value specified in the <code>returnvalue</code> argument instead of the value returned by the OLE server or DataWindow and cancel the error condition.</td>
</tr>
<tr>
<td><code>returnvalue</code></td>
<td>Any by reference. A value whose datatype matches the expected value that the OLE server would have returned. This value is used when the value of <code>action</code> is ExceptionSubstituteReturnValue!.</td>
</tr>
</tbody>
</table>
ExternalException

4 If no exception handler exists, or if the existing exception handlers do not handle the exception, the SystemError event is triggered.

5 If the SystemError event has no script, an application error occurs and the application is terminated.

Examples

Suppose your window has two instance variables: one for specifying the exception action, and another of type Any for storing a potential substitute value. Before accessing the OLE property, a script sets the instance variables to appropriate values:

```powershell
ie_action = ExceptionSubstituteReturnValue!
ia_substitute = 0
li_currentsetting = ole_1.Object.Value
```

If the command fails, a script for the ExternalException event displays the Help topic named by the OLE server, if any. It substitutes the return value you prepared and returns control to the calling script. The assignment of the substitute value to `li_currentsetting` works correctly because their datatypes are compatible:

```powershell
string ls_context

// Command line switch for WinHelp numeric context ID
ls_context = "-n " + String(helpcontext)
If Len(HelpFile) > 0 THEN
  Run("winhelp.exe " + ls_context + " " + helpfile)
END IF

action = ie_action
returnvalue = ia_substitute
```

Because the event script must serve for every automation command for the control, you need to set the instance variables to appropriate values before each automation command.

See also

Error
**FileExists**

**Description**
Occurs when a file is saved in the RichTextEdit control and the file already exists.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renfileexists</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>The name of the file</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specified in a RETURN statement):

- 0  Continue processing
- 1  Saving of document is canceled

**Usage**
The SaveDocument function can trigger the FileExists event.

**Examples**
This script for FileExists checks a flag to see if the user is performing a save (which will automatically overwrite the opened file) or wants to rename the file using Save As. For the Save As case, the script asks the user to confirm overwriting the file:

```powerScript
integer li_answer

// If user asked to Save to same file, // do not prompt for overwriting IF ib_saveas = FALSE THEN RETURN 0

li_answer = MessageBox("FileExists", & filename + " already exists. Overwrite?", & Exclamation!, YesNo!)
    MessageBox("Filename arg", filename)

// Returning a non-zero value cancels save IF li_answer = 2 THEN RETURN 1
```
**FileMessage**

**Description**
Occurs on display of a detailed information message from a MobiLink synchronization.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filemsg</td>
<td>Read-only string containing the text of the message returned from the synchronization server.</td>
</tr>
</tbody>
</table>

**Return codes**
None

**Usage**
Use this event to receive information logged by dbmlsync.

The following events can be triggered when different types of messages are sent by the synchronization server: DisplayMessage, ErrorMessage, FileMessage, and WarningMessage.

**See also**
DisplayMessage
ErrorMessage
WarningMessage

---

**Gesture**

**Description**
Occurs when an application gesture recognized by the control is completed. A gesture is a stroke or series of strokes that is recognized by the application as indicating an action. This event can only be triggered on a Tablet PC.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_inkegesture</td>
<td>InkEdit</td>
</tr>
<tr>
<td>pbm_inkpgesture</td>
<td>InkPicture</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gest</td>
<td>Integer identifying the gesture recognized. See the tables in the Usage section for argument values.</td>
</tr>
</tbody>
</table>

**Return codes**
Boolean. Return false to accept the gesture and true to ignore it.
CHAPTER 9  PowerScript Events

Usage

The Gesture event is triggered only on a Tablet PC. On a Tablet PC, the InkEdit control recognizes the following gestures that represent keystrokes that are frequently used in edit controls. To ensure that the gestures are recognized, users should draw straight lines and sharp right angles without removing the stylus from the control. InkEdit controls on other computers behave as MultiLineEdit controls and cannot accept ink input from a mouse.

On a Tablet PC, the InkPicture control recognizes the following gestures that are equivalent to mouse clicks:

<table>
<thead>
<tr>
<th>Gesture name</th>
<th>Argument value</th>
<th>Mouse action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tap</td>
<td>1</td>
<td>Left Click</td>
</tr>
<tr>
<td>Double Tap</td>
<td>2</td>
<td>Left Double Click</td>
</tr>
</tbody>
</table>

When you tap the stylus or click a mouse in an InkPicture control on a Tablet PC, the Gesture event is triggered. On other computers, a mouse click triggers the Stroke event. The CollectionMode property must be set to GestureOnly! for a double tap to be recognized. Only single-stroke gestures are recognized when CollectionMode is set to InkAndGesture!. If a gesture is not recognized, the value of the argument is 0.

Examples

This code in the Gesture event of an InkEdit control confirms to the user that the gesture was recognized:

```powerscript
CHOOSE CASE gest
CASE 0
    MessageBox("Gesture recognized", &
               "You entered a space")
CASE 1
    MessageBox("Gesture recognized", &
               "You entered a backspace")
```
GetFocus

CASE 2,4
    MessageBox("Gesture recognized", &
        "You entered a tab")
CASE 3,5
    MessageBox("Gesture recognized", &
        "You entered a return")
END CHOOSE

return false

See also
    RecognitionResult
    Stroke

GetFocus

Description
Occurs just before the control receives focus (before it is selected and becomes active).

GetFocus applies to all controls

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_bnsetfocus</td>
<td>CheckBox, CommandButton, Graph, OLE, Picture, PictureHyperLink, PictureButton, RadioButton</td>
</tr>
<tr>
<td>pbm_cbnsetfocus</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_dwnsetfocus</td>
<td>DataWindow</td>
</tr>
<tr>
<td>pbm_ensetfocus</td>
<td>SingleLineEdit, EditMask, MultiLineEdit, StaticText, StaticHyperLink</td>
</tr>
<tr>
<td>pbm_lbnsetfocus</td>
<td>ListBox, PictureListBox</td>
</tr>
<tr>
<td>pbm_lvnsetfocus</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_rensetfocus</td>
<td>RichTextEdit</td>
</tr>
<tr>
<td>pbm_sbnsetfocus</td>
<td>HScrollBar, HTrackBar, VScrollBar, VTrackBar</td>
</tr>
<tr>
<td>pbm_setfocus</td>
<td>HProgressBar, VProgressBar, DatePicker, MonthCalendar, InkEdit, InkPicture</td>
</tr>
<tr>
<td>pbm_tcnsetfocus</td>
<td>Tab</td>
</tr>
<tr>
<td>pbm_tvnsetfocus</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments
None
Return codes

Long. Return code choices (specified in a RETURN statement):

0  Continue processing

Examples

**Example 1**  This example in a SingleLineEdit control’s GetFocus event selects the text in the control when the user tabs to it:

```powerscript
This.SelectText(1, Len(This.Text))
```

**Example 2**  In Example 1, when the user clicks the SingleLineEdit rather than tabbing to it, the control gets focus and the text is highlighted, but then the click deselects the text. If you define a user event that selects the text and then post that event in the GetFocus event, the highlighting works when the user both tabs and clicks. This code is in the GetFocus event:

```powerscript
This. EVENT POST ue_select( )
```

This code is in the `ue_select` user event:

```powerscript
This.SelectText(1, Len(This.Text))
```

See also

Clicked
LoseFocus

---

**Help**

**Description**

Occurs when the user drags the question-mark button from the title bar to a menu item or a control and then clicks, or when the user clicks in a control (giving it focus) and then presses the F1 key.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_help</td>
<td>Window, Menu, DragObject</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xpos</td>
<td>Integer by value (the distance of the Help message from the left edge of the screen, in PowerBuilder units)</td>
</tr>
<tr>
<td>ypos</td>
<td>Integer by value (the distance of the Help message from the top of the screen, in PowerBuilder units)</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specified in a RETURN statement):

0  Continue processing
Usage

The question-mark button only appears in the title bar of response windows. You must set the ContextHelp property to true to enable this event.

You can script Help messages for individual menu items and controls. PowerBuilder dispatches the associated Windows message to the appropriate menu item or control.

Examples

This example codes a message box to open when the user drag-and-clicks the question-mark button over a TrackBar control:

```
MessageBox("Context Help Message", "Move the TrackBar" & 
  + " slider to change the DataWindow magnification.")
```

See also

ShowHelp

---

**Hide**

Description

Occurs just before the window is hidden.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_hidewindow</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments

None

Return codes

Long. Return code choices (specified in a RETURN statement):

- 0  Continue processing

Usage

A Hide event can occur when a sheet in an MDI frame is closed. It does not occur when closing a main, response, or pop-up window.

See also

Close
Show
**HotLinkAlarm**

**Description**
Occurs after a Dynamic Data Exchange (DDE) server application has sent new (changed) data and the client DDE application has received it.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_ddedata</td>
<td>Window</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Usage**
After establishing a hot link with a DDE server application with the StartHotLink function, actions on the server can trigger the HotLinkAlarm event.

**Examples**
This script in the HotLinkAlarm event gets information about the DDE server application and the new data:

```powerscript
string ls_data, ls_appl, ls_topic, ls_item
GetDataDDEOrigin(ls_appl, ls_topic, ls_item)
GetDataDDE(ls_data)
```

---

**Idle**

**Description**
Occurs when the Idle function has been called in an application object script and the specified number of seconds have elapsed with no mouse or keyboard activity.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Application</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
None. (Do not use a RETURN statement.)

**Examples**
This statement in an application script causes the Idle event to be triggered after 300 seconds of inactivity:

```powerscript
Idle(300)
```

In the Idle event itself, this statement closes the application:

```powerscript
HALT CLOSE
```
### InputFieldSelected

**Description**
In a RichTextEdit control, occurs when the user double-clicks an input field, allowing the user to edit the data in the field.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_reninputfieldselected</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fieldname</td>
<td>String by value (the name of the input field that was selected)</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Examples**
This script for the InputFieldSelected event of a RichTextEdit control gets the data in the input field the user is about to edit:

```powershell
going string ls_fieldvalue
ls_fieldvalue = This.InputFieldGetData(fieldname)
```

**See also**
PictureSelected

### InsertItem

**Description**
Occurs when an item is inserted in the ListView.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvninsertitem</td>
<td>ListView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>An integer that represents the index of the item being inserted into the ListView</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specified in a RETURN statement):

- 0  Continue processing
Examples

This example displays the label and index of the inserted item:

```power
ListViewItem lvi
This.GetItem(index, lvi)
sle_info.Text = "Inserted " + String(lvi.Label) & " into position " & String(index)
```

See also
DeleteItem

**ItemActivate**

**Description**

Occurs when a ListView item is clicked or double-clicked. The actual firing mechanism depends on the OneClickActivate and TwoClickActivate property settings.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnitemactivate</td>
<td>ListView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>An integer that represents the index of the item being inserted into the ListView</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0 Continue processing

**Usage**

Use the ItemActivate event instead of the Clicked or DoubleClicked event in new applications.

The following ListView property settings determine which user action fires the event:

<table>
<thead>
<tr>
<th>OneClickActivate</th>
<th>TwoClickActivate</th>
<th>Firing mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>Single click</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>Single click</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>Single click on selected item or double-click on nonselected item</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>Double-click</td>
</tr>
</tbody>
</table>
ItemChanged

Examples

This code changes a ListView item text label to uppercase lettering. The change is made in the second column of the item the user clicks or double-clicks, depending on the ListView property settings:

```
listviewitem llvi_current

This.GetItem(index, 2, llvi_current)
llvi_current.Label = Upper(llvi_current.Label)
This.SetItem(index, 2, llvi_current)
RETURN 0
```

See also

ItemChanged
ItemChanging

ItemChanged

Description

Occurs when an ListView item has changed.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnitemchanged</td>
<td>ListView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>index</code></td>
<td>The index of the item that is changing</td>
</tr>
<tr>
<td><code>focuschanged</code></td>
<td>Boolean (specifies if focus has changed for the item)</td>
</tr>
<tr>
<td><code>hasfocus</code></td>
<td>Boolean (specifies whether the item has focus)</td>
</tr>
<tr>
<td><code>selectionchange</code></td>
<td>Boolean (specifies whether the selection has changed for the item)</td>
</tr>
<tr>
<td><code>selected</code></td>
<td>Boolean (specifies whether the item is selected)</td>
</tr>
<tr>
<td><code>otherchange</code></td>
<td>Boolean (specifies if anything other than focus or selection has changed for the item)</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>
Examples
This example checks whether the event is occurring because focus has changed to the item:

```powerScript
ListViewItem l_lvi
lv_list.GetItem(index, l_lvi)
IF focuschange and hasfocus THEN
    sle1.Text = String(lvi.label) + " has focus."
END IF
```

See also ItemChanged in the DataWindow Reference or the online Help

### ItemChanging

**Description**
Occurs just before a ListView changes.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnitemchanging</td>
<td>ListView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the item that has changed</td>
</tr>
<tr>
<td>focuschange</td>
<td>Boolean (specifies if focus is changing for the item)</td>
</tr>
<tr>
<td>hasfocus</td>
<td>Boolean (specifies whether the item has focus)</td>
</tr>
<tr>
<td>selectionchange</td>
<td>Boolean (specifies whether the selection is changing for the item)</td>
</tr>
<tr>
<td>selected</td>
<td>Boolean (specifies whether the item is selected)</td>
</tr>
<tr>
<td>otherchange</td>
<td>Boolean (specifies if anything other than focus or selection has changed for the item)</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>

See also ItemChanged
**ItemCollapsed**

**Description**
Occurs when a TreeView item has collapsed.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnitemcollapsed</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by reference (the handle of the collapsed TreeViewItem)</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specified in a RETURN statement):

- 0  Continue processing

**Examples**

This example changes the picture for the collapsed item:

```powerbuilder
TreeViewItem l_tvi
integer li_level

This.GetItem(handle, l_tvi)

CHOOSE CASE l_tvi.Level
CASE 1
   l_tvi.PictureIndex = 1
   l_tvi.SelectedPictureIndex = 1
CASE 2
   l_tvi.PictureIndex = 2
   l_tvi.SelectedPictureIndex = 2
CASE 3
   l_tvi.PictureIndex = 3
   l_tvi.SelectedPictureIndex = 3
CASE 4
   l_tvi.PictureIndex = 4
   l_tvi.SelectedPictureIndex = 4
END CHOOSE
This.SetItem(handle, l_tvi)
```

**See also**
ItemCollapsing
**ItemCollapsing**

**Description**
Occurs when a TreeView item is collapsing.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnitemcollapsing</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by reference (the handle of the collapsing item)</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Usage**
The ItemCollapsing event occurs before the ItemCollapsed event.

**Examples**
This example changes the picture for the collapsing item:

```powerlin
TreeViewItem l_tvi
integer li_level

This.GetItem(handle, l_vti)

CHOOSE CASE l_tvi.level
CASE 1
    l_tvi.PictureIndex = 1
    l_tvi.SelectedPictureIndex = 1
CASE 2
    l_tvi.PictureIndex = 2
    l_tvi.SelectedPictureIndex = 2
CASE 3
    l_tvi.PictureIndex = 3
    l_tvi.SelectedPictureIndex = 3
CASE 4
    l_tvi.PictureIndex = 4
    l_tvi.SelectedPictureIndex = 4
END CHOOSE

This.SetItem(handle, l_tvi)
```

**See also**
ItemCollapsed
**ItemExpanded**

**Description**
Occurs when a TreeView item has expanded.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnitemexpanded</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by reference (the handle of the expanded item)</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Usage**
The ItemExpanded event occurs after the ItemExpanding event.

**Examples**
This example sets the picture and selected picture for the expanded item:

```powerbuilder
TreeViewItem l_tvi
integer li_level

This.GetItem(handle, l_tvi)

CHOOSE CASE l_tvi.Level
CASE 1
  l_tvi.PictureIndex = 5
  l_tvi.SelectedPictureIndex = 1
CASE 2
  l_tvi.PictureIndex = 5
  l_tvi.SelectedPictureIndex = 2
CASE 3
  l_tvi.PictureIndex = 5
  l_tvi.SelectedPictureIndex = 3
CASE 4
  l_tvi.PictureIndex = 4
  l_tvi.SelectedPictureIndex = 5
END CHOOSE
This.SetItem(handle, l_tvi)
```

**See also**
ItemExpanding

---

256

PowerBuilder
## ItemExpanding

### Description

Occurs while a TreeView item is expanding.

### Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvntitemexpanding</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by reference (the handle of the expanding TreeView item)</td>
</tr>
</tbody>
</table>

### Return codes

Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing
- 1  Prevents the TreeView from expanding

### Usage

The ItemExpanding event occurs before the ItemExpanded event.

### Examples

This example sets the picture and selected picture for the expanding item:

```powerShell
TreeViewItem l_tvi
integer li_level

This.GetItem(handle, l_tvi)

CHOOSE CASE l_tvi.Level
    CASE 1
        l_tvi.PictureIndex = 5
        l_tvi.SelectedPictureIndex = 1
    CASE 2
        l_tvi.PictureIndex = 5
        l_tvi.SelectedPictureIndex = 2
    CASE 3
        l_tvi.PictureIndex = 5
        l_tvi.SelectedPictureIndex = 3
    CASE 4
        l_tvi.PictureIndex = 4
        l_tvi.SelectedPictureIndex = 5
END CHOOSE

This.SetItem(handle, l_tvi)
```

### See also

ItemExpanded
ItemPopulate

Occurs when a TreeView item is being populated with children.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnitempopulate</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by reference (the handle of the TreeView item being populated)</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specified in a RETURN statement):

- 0  Continue processing

**Examples**

This example displays the name of the TreeView item you are populating in a SingleLineEdit:

```powerbuilder
TreeViewItem tvi
This.GetItem(handle, tvi)
sle_get.Text = "Populating TreeView item " &
    + String(tvi.Label) + " with children"
```

**See also**

ItemExpanding

---

**Key**

Occurs when the user presses a key.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnkeydown</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_renkey</td>
<td>RichTextEdit</td>
</tr>
<tr>
<td>pbm_tcnkeydown</td>
<td>Tab</td>
</tr>
<tr>
<td>pbm_tvnkeydown</td>
<td>TreeView</td>
</tr>
<tr>
<td>pbm_keydown</td>
<td>Window</td>
</tr>
</tbody>
</table>
CHAPTER 9  PowerScript Events

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>key</em></td>
<td>KeyCode by value. A value of the KeyCode enumerated datatype indicating the key that was pressed (for example, KeyA! or KeyF1!).</td>
</tr>
</tbody>
</table>
| *keyflags* | UnsignedLong by value (the modifier keys that were pressed with the key). Values are:  
  1 Shift key  
  2 Ctrl key  
  3 Shift and Ctrl keys |

Return codes

Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing
- 1  Do not process the key (RichTextEdit controls only)

Usage

Some PowerBuilder controls capture keystrokes so that the window is prevented from getting a Key event. These include ListView, TreeView, Tab, RichTextEdit, and the DataWindow edit control. When these controls have focus you can respond to keystrokes by writing a script for an event for the control. If there is no predefined event for keystrokes, you can define a user event and associate it with a pbm code.

For a RichTextEdit control, pressing a key can perform document formatting. For example, Ctrl+b applies bold formatting to the selection. If you specify a return value of 1, the document formatting associated with the key will not be performed.

If the user presses a modifier key and holds it down while pressing another key, the Key event occurs twice: once when the modifier key is pressed and again when the second key is pressed. If the user releases the modifier key before pressing the second key, the value of *keyflags* will change in the second occurrence.

When the user releases a key, the Key event does not occur. Therefore, if the user releases a modifier key, you do not know the current state of the modifier keys until another key is pressed.

Examples

This example causes a beep when the user presses F1 or F2, as long as Shift and Ctrl are not pressed:

```powerbuilder
IF keyflags = 0 THEN
  IF key = KeyF1! THEN
    Beep(1)
  ELSEIF key = KeyF2! THEN
    Beep(20)
```
This line displays the value of keyflags when a key is pressed.

\[ \text{st}_1.\text{Text} = \text{String(keyflags)} \]

See also SystemKey

## LineDown

**Description**

Occurs when the user clicks the down arrow of the vertical scroll bar or presses the down arrow on the keyboard when the focus is on a track bar.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_sbnlinedown</td>
<td>VScrollBar, VTrackBar</td>
</tr>
</tbody>
</table>

**Arguments**

None

**Return codes**

Long. Return code choices (specify in a RETURN statement):

0 Continue processing

**Usage**

When the user clicks in a vertical scroll bar or presses the down arrow key with focus in a vertical track bar, nothing happens unless you have scripts that change the bar’s Position property. For the scroll bar arrows and arrow keys for the track bar, use the LineUp and LineDown events; for clicks in the scroll bar or track bar background above and below the thumb, use the PageUp and PageDown event; for dragging the thumb itself, use the Moved event.

**Examples**

This code in the LineDown event causes the thumb to move down when the user clicks on the down arrow of the vertical scroll bar and displays the resulting position in the StaticText control \( \text{st}_1 \):

\[
\text{IF This.Position > This.MaxPosition - 1 THEN} \\
\quad \text{This.Position = MaxPosition} \\
\text{ELSE} \\
\quad \text{This.Position = This.Position + 1} \\
\text{END IF}
\]

\[ \text{st}_1.\text{Text} = \"LineDown \" + \text{String(This.Position)} \]

See also LineLeft, LineRight, LineUp, PageDown

---

260 PowerBuilder
**LineLeft**

Description

Occurs when the user clicks in the left arrow of the horizontal scroll bar or presses the left arrow key on the keyboard when focus is on a horizontal track bar.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_sbnlineup</td>
<td>HScrollBar, HTrackBar</td>
</tr>
</tbody>
</table>

Arguments

None

Return codes

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage

When the user clicks in a horizontal scroll bar or presses the left arrow key on the keyboard in a horizontal track bar, nothing happens unless you have scripts that change the bar’s Position property. For the scroll bar arrows and left arrow keys in a track bar, use the LineLeft and LineRight events; for clicks in the background above and below the thumb, use the PageLeft and Right events; for dragging the thumb itself, use the Moved event.

Examples

This code in the LineLeft event causes the thumb to move left when the user clicks on the left arrow of the horizontal scroll bar and displays the resulting position in the StaticText control st_1:

```powerscript
IF This.Position < This.MinPosition + 1 THEN
  This.Position = MinPosition
ELSE
  This.Position = This.Position - 1
END IF

st_1.Text = "LineLeft " + String(This.Position)
```

See also

LineDown
LineRight
LineUp
PageLeft
LineRight

Description
Occurs when the user clicks in the right arrow of the horizontal scroll bar or presses the right arrow key on the keyboard when focus is on a horizontal track bar.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_sbnlinedown</td>
<td>HScrollBar, HTrackBar</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage
When the user clicks in a horizontal scroll bar or presses the right arrow key on the keyboard in a horizontal track bar, nothing happens unless you have scripts that change the bar’s Position property. For the scroll bar arrows and arrow keys in a track bar, use the LineLeft and LineRight events; for clicks in the background above and below the thumb, use the PageLeft and Right events; for dragging the thumb itself, use the Moved event.

Examples
This code in the LineRight event causes the thumb to move right when the user clicks on the right arrow of the horizontal scroll bar and displays the resulting position in the StaticText control st_1:

```plaintext
IF This.Position > This.MaxPosition - 1 THEN
   This.Position = MaxPosition
ELSE
   This.Position = This.Position + 1
END IF

st_1.Text = "LineRight " + String(This.Position)
```

See also
LineDown
LineLeft
LineUp
PageRight
**LineUp**

**Description**
Occurs when the user clicks the up arrow of the vertical scroll bar or presses the up arrow on the keyboard when the focus is on a track bar.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_sbnlineup</td>
<td>VScrollBar, VTrackBar</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code choices (specify in a RETURN statement):
- 0  Continue processing

**Usage**
When the user clicks in a vertical scroll bar or presses the up arrow key with focus in a vertical track bar, nothing happens unless you have scripts that change the bar’s Position property. For the scroll bar arrows and arrow keys for the track bar, use the LineUp and LineDown events; for clicks in the scroll bar or track bar background above and below the thumb, use the PageUp and PageDown event; for dragging the thumb itself, use the Moved event.

**Examples**
This code in the LineUp event causes the thumb to move up when the user clicks on the up arrow of the vertical scroll bar and displays the resulting position in the StaticText control st_1:

```powershell
IF This.Position < This.MinPosition + 1 THEN
    This.Position = MinPosition
ELSE
    This.Position = This.Position - 1
END IF

st_1.Text = "LineUp " + String(This.Position)
```

**See also**
LineDown
LineLeft
LineRight
PageUp
**LoseFocus**

**Description**
Occurs just before a control loses focus (before it becomes inactive).

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_controltypekillfocus</td>
<td>UserObject (standard visual user objects only)</td>
</tr>
<tr>
<td>pbm_bnkillfocus</td>
<td>CheckBox, CommandButton, Graph, OLE, Picture, PictureHyperLink, PictureButton, RadioButton, StaticText, StaticHyperLink</td>
</tr>
<tr>
<td>pbm_cbnkillfocus</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_dwnkillfocus</td>
<td>DataWindow</td>
</tr>
<tr>
<td>pbm_enkillfocus</td>
<td>SingleLineEdit, EditMask, MultiLineEdit</td>
</tr>
<tr>
<td>pbm_killfocus</td>
<td>HPProgressBar, VProgressBar, DatePicker, MonthCalendar, InkEdit, InkPicture</td>
</tr>
<tr>
<td>pbm_lbnkillfocus</td>
<td>ListBox, PictureListBox</td>
</tr>
<tr>
<td>pbm_lvnkillfocus</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_renkillfocus</td>
<td>RichTextEdit</td>
</tr>
<tr>
<td>pbm_sbnkillfocus</td>
<td>HScrollBar, HTrackBar, VScrollBar, VTrackBar</td>
</tr>
<tr>
<td>pbm_tcnkillfocus</td>
<td>Tab</td>
</tr>
<tr>
<td>pbm_tvnnkillfocus</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code choices (specify in a RETURN statement):

0  Continue processing

**Usage**
Write a script for a control’s LoseFocus event if you want some processing to occur when the user changes focus to another control.

For controls that contain editable text, losing focus can also cause a Modified event to occur.

In a RichTextEdit control, a LoseFocus event occurs when the user clicks on the control’s toolbar. The control does not actually lose focus.

Because the MessageBox function 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.

**Examples**

**Example 1**  In this script for the LoseFocus event of a SingleLineEdit sle_town, the user is reminded to enter information if the text box is left empty:

```pascal
IF sle_town.Text = "" THEN
   st_status.Text = "You have not specified a town."
END IF
```
Example 2  Statements in the LoseFocus event for a DataWindow control `dw_emp` can trigger a user event whose script validates the last item the user entered.

This statement triggers the user event `ue_accept`:

```
dw_emp.EVENT ue_accept( )
```

This statement in `ue_accept` calls the `AcceptText` function:

```
dw_emp.AcceptText( )
```

This script for the LoseFocus event of a RichTextEdit control performs processing when the control actually loses focus:

```
GraphicObject l_control

// Check whether the RichTextEdit still has focus
l_control = GetFocus()
IF TypeOf(l_control) = RichTextEdit! THEN RETURN 0

// Perform processing only if RichTextEdit lost focus
...
```

This script gets the name of the control instead:

```
GraphicObject l_control
string ls_name
l_control = GetFocus()
ls_name = l_control.Classname( )
```

See also  GetFocus

Modified

<table>
<thead>
<tr>
<th>Description</th>
<th>Occurs when the contents in the control have changed.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_cbnmodified</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_enmodified</td>
<td>SingleLineEdit, EditMask, MultiLineEdit</td>
</tr>
<tr>
<td>pbm_inkemodified</td>
<td>InkEdit</td>
</tr>
<tr>
<td>pbm_renmodified</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

Arguments  None
Modified

Return codes
Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage
For plain text controls, the Modified event occurs when the user indicates being finished by pressing Enter or tabbing away from the control.

For InkEdit and RichText Edit controls, the value of the Modified property controls the Modified event. If the property is false, the event occurs when the first change occurs to the contents of the control. The change also causes the property to be set to true, which suppresses the Modified event. You can restart checking for changes by setting the property back to false.

Resetting the Modified property is useful when you insert text or a document in the control, which triggers the event and sets the property (it is reporting the change to the control’s contents). To find out when the user begins making changes to the content, set the Modified property back to false in the script that opens the document. When the user begins editing, the property will be reset to true and the event will occur again.

A Modified event can be followed by a LoseFocus event.

Examples
In this example, code in the Modified event performs validation on the text the user entered in a SingleLineEdit control sle_color. If the user did not enter RED, WHITE, or BLUE, a message box indicates what is valid input; for valid input, the color of the text changes:

```powerbuilder
string ls_color

This.BackColor = RGB(150,150,150)
ls_color = Upper(This.Text)
CHOOSE CASE ls_color
CASE "RED"
   This.TextColor = RGB(255,0,0)
CASE "BLUE"
   This.TextColor = RGB(0,0,255)
CASE "WHITE"
   This.TextColor = RGB(255,255,255)
CASE ELSE
   This.Text = ""
   MessageBox("Invalid input", &
   "Enter RED, WHITE, or BLUE.")
END CHOOSE
```

This is not a realistic example: user input of three specific choices is more suited to a list box; in a real situation, the allowed input might be more general.

See also
LoseFocus
MouseDown

The MouseDown event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>RichTextEdit control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Window</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1  
For RichTextEdit controls

Description
Occurs when the user presses the left mouse button on the RichTextEdit control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renlbuttondown</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Examples
This code in a RichTextEdit control’s MouseDown event assigns text to the SingleLineEdit sle_1 when the user presses the left mouse button:

```
sle_1.text = "Mouse Down"
```

See also
Clicked
MouseMove
MouseUp

Syntax 2  
For windows

Description
Occurs when the user presses the left mouse button in an unoccupied area of the window (any area with no visible, enabled object).

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lbuttondown</td>
<td>Window</td>
</tr>
</tbody>
</table>
MouseDown

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| flags    | UnsignedLong by value (the modifier keys and mouse buttons that are pressed). Values are:  
- 1 – Left mouse button  
- 2 – Right mouse button  
- 4 – Shift key  
- 8 – Ctrl key  
- 16 – Middle mouse button  
In the MouseDown event, the left mouse button is always down, so 1 is always summed in the value of flags. For an explanation of flags, see Syntax 2 of MouseMove on page 269. |
| xpos     | Integer by value (the distance of the pointer from the left edge of the window’s workspace in pixels). |
| ypos     | Integer by value (the distance of the pointer from the top of the window’s workspace in pixels). |

Return codes

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
</table>
| Long        | Return code choices (specify in a RETURN statement):  
0 Continue processing |

Examples

**Example 1**  This code in the MouseDown event displays the window coordinates of the pointer as reported in the xpos and ypos arguments:

```powershell
sle_2.Text = "Position of Pointer is: " + &
String(xpos) + "," + String(ypos)
```

**Example 2**  This code in the MouseDown event checks the value of the flags argument, and reports which modifier keys are pressed in the SingleLineEdit sle_mkey:

```powershell
CHOOSE CASE flags
CASE 1
    sle_mkey.Text = "No modifier keys pressed"
CASE 5
    sle_mkey.Text = "SHIFT key pressed"
CASE 9
    sle_mkey.Text = "CONTROL key pressed"
CASE 13
    sle_mkey.Text = "SHIFT and CONTROL keys pressed"
END CHOOSE
```

See also

Clicked
MouseMove
MouseUp
# MouseMove

The MouseMove event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>RichTextEdit control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Window</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

## Syntax 1  
**For RichTextEdit controls**

- **Description**: Occurs when the mouse has moved within the RichTextEdit control.

- **Event ID**
  - `pbm_renmousemove` for RichTextEdit

- **Arguments**: None

- **Return codes**: Long. Return code choices (specify in a RETURN statement):
  - 0 Continue processing

- **See also**: Clicked, MouseDown, MouseUp

## Syntax 2  
**For windows**

- **Description**: Occurs when the pointer is moved within the window.

- **Event ID**
  - `pbm_mousemove` for window

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbm_mousemove</code></td>
<td>Window</td>
</tr>
</tbody>
</table>
MouseMove

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| flags    | UnsignedLong by value (the modifier keys and mouse buttons that are pressed). Values are:  
|          | • 1 – Left mouse button  
|          | • 2 – Right mouse button  
|          | • 4 – Shift key  
|          | • 8 – Ctrl key  
|          | • 16 – Middle mouse button  
|          | Flags is the sum of all the buttons and keys that are pressed. |

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xpos</td>
<td>Integer by value (the distance of the pointer from the left edge of the window’s workspace in pixels).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ypos</td>
<td>Integer by value (the distance of the pointer from the top of the window’s workspace in pixels).</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>

Usage

Because flags is a sum of button and key numbers, you can find out what keys are pressed by subtracting the largest values one by one and checking the value that remains. For example:

• If flags is 5, the Shift key (4) and the left mouse button (1) are pressed.

• If flags is 14, the Ctrl key (8), the Shift key (4), and the right mouse button (2) are pressed.

This code handles all the buttons and keys (the local boolean variables are initialized to false by default):

```powerbuilder
boolean lb_left_button, lb_right_button
boolean lb_middle_button, lb_Shift_key, lb_control_key
integer li_flags

li_flags = flags
IF li_flags > 15 THEN
    // Middle button is pressed
    lb_middle_button = TRUE
    li_flags = li_flags - 16
END IF
```
IF li_flags 7 THEN
   // Control key is pressed
   lb_control_key = TRUE
   li_flags = li_flags - 8
END IF

IF li_flags > 3 THEN
   // Shift key is pressed
   lb_Shift_key = TRUE
   li_flags = li_flags - 4
END IF

IF li_flags > 1 THEN
   // Right button is pressed
   lb_right_button = TRUE
   li_flags = li_flags - 2
END IF

IF li_flags = 1 THEN lb_left_button = TRUE

Most controls in a window do not capture MouseMove events—the
MouseMove event is not mapped by default. If you want the window’s
MouseMove event to be triggered when the mouse moves over a control, you
must map a user-defined event to the pbm_mousedown event for the control.
The following code in the control’s user-defined MouseMove event triggers
the window’s MouseMove event:

   Parent.EVENT MouseMove(0, Parent.PointerX(),
         Parent.PointerY())

Examples

This code in the MouseMove event causes a meter OLE custom control to rise
and fall continually as the mouse pointer is moved up and down in the window
workspace:

   This.uf_setmonitor(ypos, ole_verticalmeter, &
          This.WorkspaceHeight() )

Uf_setmonitor is a window function that scales the pixels to the range of the
gauge. It accepts three arguments: the vertical position of the mouse pointer, an
OLECustomControl reference, and the maximum range of the mouse pointer
for scaling purposes:

   double ld_gaugemax, ld_gaugemin
   double ld_gaugerange, ld_value
MouseUp

// Ranges for monitor-type control
ld_gaugemax = ocxitem.Object.MaxValue
ld_gaugemin = ocxitem.Object.MinValue
ld_gaugerange = ld_gaugemax - ld_gaugemin

// Horizontal position of mouse within window
ld_value = data * ld_gaugerange / range + ld_gaugemin

// Set gauge
ocxitem.Object.Value = Round(ld_value, 0)

RETURN 1

The OLE custom control also has a MouseMove event. This code in that event keeps the gauge responding when the pointer is over the gauge. (You need to pass values for the arguments that are usually handled by the system; the mouse position values are specified in relation to the parent window.) For example:

Parent.EVENT MouseMove(0, Parent.PointerX(), & Parent.PointerY())

See also
Clicked
MouseDown
MouseUp

MouseUp

The MouseUp event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>RichTextEdit control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Window</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1 For RichTextEdit controls

Description
Occurs when the user releases the left mouse button in a RichTextEdit control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renlbuttonup</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>
Arguments: None

Return codes: Long. Return code choices (specify in a RETURN statement):
- 0 Continue processing

Usage: A Clicked event also occurs when the mouse button is released.

Examples: The following code in a RichTextEdit control’s MouseUp event assigns text to the SingleLineEdit sle_1 when the user releases the left mouse button:

```plaintext
sle_1.Text = "Mouse Up"
```

See also: Clicked, MouseDown, MouseMove

**Syntax 2**

For windows

Description: Occurs when the user releases the left mouse button in an unoccupied area of the window (any area with no visible enabled object).

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lbuttonup</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| flags    | UnsignedLong by value (the modifier keys and mouse buttons that are pressed). Values are:
  - 1 – Left mouse button
  - 2 – Right mouse button
  - 4 – Shift key
  - 8 – Ctrl key
  - 16 – Middle mouse button
In the MouseUp event, the left mouse button is being released, so 1 is not summed in the value of flags. For an explanation of flags, see Syntax 2 of MouseMove on page 269.
| xpos     | Integer by value (the distance of the pointer from the left edge of the window’s workspace in pixels). |
| ypos     | Integer by value (the distance of the pointer from the top of the window’s workspace in pixels). |
MouseUp

Return codes

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage

A Clicked event also occurs when the mouse button is released.

Examples

Example 1  This code in the window’s MouseUp event displays in the SingleLineEdit sle_2 the window coordinates of the pointer when the button is released as reported in the xpos and ypos arguments.

\[
\text{sle}_2.\text{Text} = "\text{Position of Pointer is: }" + \& \\
\text{String(xpos)} + "," + \text{String(ypos)}
\]

Example 2  This code in the window’s MouseUp event checks the value of the flags argument and reports which modifier keys are pressed in the SingleLineEdit sle_mkey.

\[
\text{CHOOSE CASE flags} \\
\text{CASE 0} \\
\hspace{1em} \text{sle}_m\text{key}.\text{Text} = "\text{No modifier keys pressed}\"
\]

\[
\text{CASE 4} \\
\hspace{1em} \text{sle}_m\text{key}.\text{Text} = "\text{SHIFT key pressed}\"
\]

\[
\text{CASE 8} \\
\hspace{1em} \text{sle}_m\text{key}.\text{Text} = "\text{CONTROL key pressed}\"
\]

\[
\text{CASE 12} \\
\hspace{1em} \text{sle}_m\text{key}.\text{Text} = "\text{SHIFT and CONTROL keys pressed}\"
\]

END CHOOSE

See also

Clicked
MouseDown
MouseMove
### Moved

**Description**

Occurs when the user moves the scroll box, either by clicking on the arrows or by dragging the box itself.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_sbnthumbtrack</td>
<td>HScrollBar, HTrackBar, VScrollBar, VTrackBar</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scrollpos</td>
<td>Integer by value (a number indicating position of the scroll box within the range of values specified by the MinPosition and MaxPosition properties)</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Usage**

The Moved event updates the Position property of the scroll bar with the value of `scrollpos`.

**Examples**

This statement in the Moved event displays the new position of the scroll box in a StaticText control:

```
st_1.Text = "Moved " + String(scrollpos)
```

**See also**

LineDown
LineLeft
LineRight
LineUp
PageDown
PageLeft
PageRight
PageUp
**Notify**

**Description**

Occurs when a TreeView control sends a WM_NOTIFY message to its parent.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_notify</td>
<td>TreeView controls</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wparam</td>
<td>UnsignedLong by value containing the ID of the control sending the message. This value is not guaranteed to be unique.</td>
</tr>
<tr>
<td>lparam</td>
<td>Long by value containing a pointer to a structure that contains the window handle and identifier of the control sending a message and a notification code.</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

**Usage**

The lparam argument can point to an NMHDR structure or to a larger structure that contains an NMHDR structure as its first member. Since the wparam value is not guaranteed to be unique, you should use the identifier in the NMHDR structure.

You can use this event to process custom drawing messages.

---

**Open**

The Open event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Window</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

For the application object

**Description**

Occurs when the user starts the application.
Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Application</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commandline</td>
<td>String by value. Additional arguments are included on the command line after the name of the executable program.</td>
</tr>
</tbody>
</table>

Return codes

None (do not use a RETURN statement)

Usage

This event can establish database connection parameters and open the main window of the application.

On Windows

You can specify command line arguments when you use the Run command from the Start menu or as part of the Target specification when you define a shortcut for your application.

There is no way to specify command line values when you are testing your application in the development environment.

In other events and functions, you can call the CommandParm function to get the command line arguments.

For an example of parsing the string in commandline, see CommandParm on page 387.

Examples

This example populates the SQLCA global variable from the application’s initialization file, connects to the database, and opens the main window:

```powershell
/* Populate SQLCA from current myapp.ini settings */
SQLCA.DBMS = ProfileString("myapp.ini", "database", & "dbms", "]")
SQLCA.Database = ProfileString("myapp.ini", & "database", "database", "]")
SQLCA.UserID = ProfileString("myapp.ini", "database", & "userid", "]")
SQLCA.DBPass = ProfileString("myapp.ini", "database", & "dbpass", "]")
SQLCA.Logid = ProfileString("myapp.ini", "database", & "logid", "]")
SQLCA.Logpass = ProfileString("myapp.ini", & "database", "LogPassWord", "]")
SQLCA.Servername = ProfileString("myapp.ini", & "database", "servername", "]")
```
SQLCA.DBParm = ProfileString("myapp.ini", "database", &
   "dbparm", ")

CONNECT;

IF SQLCA.Sqlcode <> 0 THEN
   MessageBox("Cannot Connect to Database", &
             SQLCA.SQLErrText)
   RETURN
END IF

/* Open MDI frame window */
Open(w_genapp_frame)

See also Close

Syntax 2

For windows

Occurs when a window is opened by one of the Open functions. The event
occurs after the window has been opened but before it is displayed.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_open</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments

None

Return codes

Long. Return code choices (specify in a RETURN statement):

0 Continue processing

Usage

These functions trigger the Open event:

Open
OpenWithParm
OpenSheet
OpenSheetWithParm

When the Open event occurs, the controls on the window already exist (their Constructor events have occurred). In the Open event script, you can refer to objects in the window and affect their appearance or content. For example, you can disable a button or retrieve data for a DataWindow.

Some actions are not appropriate in the Open event, even though all the controls exist. For example, calling the SetRedraw function for a control fails because the window is not yet visible.
Closing a window by calling the `Close` function in any of the window’s events or in an event of any control on the window can cause PowerBuilder to crash if the `Close` function is not the last statement in the event script. You can avoid this issue by calling the `Close` function in the last statement of the event script, or in a user-defined event that is posted from the event script. For example, the following code in the Open event script for a window called `w_1` can cause a crash:

```powerbuilder
// w_1 Open event script
close(this)
open(w_2) // causes crash
```

This code does not cause a crash:

```powerbuilder
// w_1 ue_postopen event script
close(this)
```

```powerbuilder
// w_1 Open event script
open(w_2)
this.Post Event ue_postopen()
```

### Changing the WindowState property
Do not change the `WindowState` property in the Open event of a window opened as a sheet. Doing so might result in duplicate controls on the title bar. You can change the property in other scripts once the window is open.

When a window is opened, other events occur, such as Constructor for each control in the window, Activate and Show for the window, and GetFocus for the first control in the window’s tab order.

When a sheet is opened in an MDI frame, other events occur, such as Show and Activate for the sheet and Activate for the frame.

#### Examples
When the window contains a DataWindow control, you can retrieve data for it in the Open event. In this example, values for the transaction object SQLCA have already been set up:

```powerbuilder
dw_1.SetTransObject(SQLCA)
dw_1.Retrieve()
```

#### See also
Activate
Constructor
Show
Other

Description
Occurs when a system message occurs that is not a PowerBuilder message.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_other</td>
<td>Windows and controls that can be placed in windows</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wparam</td>
<td>UnsignedLong by value</td>
</tr>
<tr>
<td>lparam</td>
<td>Long by value</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage
The Other event is no longer useful, because you can define your own user events. You should avoid using it, because it slows performance while it checks every Windows message.

PageDown

Description
Occurs when the user clicks in the open space below the scroll box.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_sbnpagedown</td>
<td>VScrollBar, VTrackBar</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage
When the user clicks in a vertical scroll bar, nothing happens unless you have scripts that change the scroll bar’s Position property. For the scroll bar arrows, use the LineUp and LineDown events; for clicks in the scroll bar background above and below the thumb, use the PageUp and PageDown events; for dragging the thumb itself, use the Moved event.

Examples

**Example 1**  This code in the VScrollBar’s PageDown event uses a predetermined paging value stored in the instance variable `ii_pagesize` to change the position of the scroll box (you would need additional code to change the view of associated controls according to the scroll bar position):
IF This.Position > &
  This.MaxPosition - ii_pagesize THEN
  This.Position = MaxPosition
ELSE
  This.Position = This.Position + ii_pagesize
END IF
RETURN 0

Example 2  This example changes the position of the scroll box by a predetermined page size stored in the instance variable ii_pagesize and scrolls forward through a DataWindow control 10 rows for each page:

long ll_currow, ll_nextrow

This.Position = This.Position + ii_pagesize
ll_currow = dw_1.GetRow()
ll_nextrow = ll_currow + 10
dw_1.ScrollToRow(ll_nextrow)
dw_1.SetRow(ll_nextrow)

See also  LineDown
           PageLeft
           PageRight
           PageUp

PageLeft

Description  Occurs when the open space to the left of the scroll box is clicked.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_sbnpageup</td>
<td>HScrollBar, HTrackBar</td>
</tr>
</tbody>
</table>

Arguments  None

Return codes  Long. Return code choices (specify in a RETURN statement):

| 0  | Continue processing |

Usage  When the user clicks in a horizontal scroll bar, nothing happens unless you have scripts that change the scroll bar’s Position property. For the scroll bar arrows, use the LineLeft and LineRight events; for clicks in the scroll bar background above and below the thumb, use the PageLeft and Right events; for dragging the thumb itself, use the Moved event.
**PageLeft**

**Examples**

This code in the PageLeft event causes the thumb to move left a predetermined page size when the user clicks on the left arrow of the horizontal scroll bar (the page size is stored in the instance variable `ii_pagesize`):

```
IF This.Position < &
This.MinPosition + ii_pagesize THEN
    This.Position = MinPosition
ELSE
    This.Position = This.Position - ii_pagesize
END IF
```

**See also**

LineLeft  
PageDown  
PageRight  
PageUp
PageRight

Description
Occurs when the open space to the right of the scroll box is clicked.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_sbnpagedown</td>
<td>HScrollBar, HTrackBar</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

Usage
When the user clicks in a horizontal scroll bar, nothing happens unless you have scripts that change the scroll bar’s Position property:

- For the scroll bar arrows, use the LineLeft and LineRight events.
- For clicks in the scroll bar background above and below the thumb, use the PageLeft and Right event.
- For dragging the thumb itself, use the Moved event.

Examples
This code in the PageRight event causes the thumb to move right when the user clicks on the right arrow of the horizontal scroll bar (the page size is stored in the instance variable ii_pagesize):

```powerscript
IF This.Position > &
    This.MaxPosition - ii_pagesize THEN
    This.Position = MaxPosition
ELSE
    This.Position = This.Position + ii_pagesize
END IF
```

See also
LineRight
PageDown
PageLeft
PageUp
PageUp

Description
Occurs when the user clicks in the open space above the scroll box (also called the thumb).

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_sbnpageup</td>
<td>VScrollBar, VTrackBar</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):
0 Continue processing

Usage
When the user clicks in a vertical scroll bar, nothing happens unless you have scripts that change the scroll bar’s Position property:

- For the scroll bar arrows, use the LineUp and LineDown events.
- For clicks in the scroll bar background above and below the thumb, use the PageUp and PageDown events.
- For dragging the thumb itself, use the Moved event.

Examples

**Example 1** This code in the PageUp event causes the thumb to move up when the user clicks on the up arrow of the vertical scroll bar (the page size is stored in the instance variable `ii_pagesize`):

```powerbuilder
IF This.Position < &
   This.MinPosition + ii_pagesize THEN
   This.Position = MinPosition
ELSE
   This.Position = This.Position - ii_pagesize
END IF
```

**Example 2** This example changes the position of the scroll box by a predetermined page size stored in the instance variable `ii_pagesize` and scrolls backwards through a DataWindow control 10 rows for each page:

```powerbuilder
long ll_currow, ll_prevrow
This.Position = This.Position - ii_pagesize
ll_currow = dw_1.GetRow() 
ll_prevrow = ll_currow - 10
dw_1.ScrollToRow(ll_prevrow)
dw_1.SetRow(ll_prevrow)
```

See also
LineUp
PageDown
PageLeft
PageRight
**PictureSelected**

**Description**
Occurs when the user selects a picture in the RichTextEdit control by clicking it.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renpictureselected</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Examples**
When the user clicks a picture in a RichTextEdit control `rte_1`, the picture is selected. This code for the PictureSelected event selects the rest of the contents, copies the contents to a string with RTF formatting intact, and pastes the formatted text into a second RichTextEdit `rte_2`:

```power_script
string ls_transfer_rtf
This.SelectTextAll()
ls_transfer_rtf = This.CopyRTF()
rte_2.PasteRTF(ls_transfer_rtf)
```

**See also**
InputFieldSelected

**PipeEnd**

**Description**
Occurs when pipeline processing is completed.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_pipeend</td>
<td>Pipeline</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Usage**
You can use the PipeEnd event to check the status of pipeline processing. The Start and Repair functions initiate pipeline processing.
PipeMeter

For a complete example of using a Pipeline object, see Application Techniques.

Examples

This code in a Pipeline user object’s PipeEnd event reports pipeline status in a StaticText control:

```plaintext
ist_status.Text = "Finished Pipeline Execution ..."
```

See also

PipeMeter
PipeStart

PipeMeter

Description

Occurs during pipeline processing after each block of rows is read or written. The Commit factor specified for the Pipeline in the Pipeline painter determines the size of each block.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_pipemeter</td>
<td>Pipeline</td>
</tr>
</tbody>
</table>

Arguments

None

Return codes

Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

Usage

The Start and Repair functions initiate pipeline processing.

In the Pipeline painter, you can specify a Commit factor specifying the number of rows that will be transferred before they are committed to the database. The PipeMeter event occurs for each block of rows as specified by the Commit factor.

For a complete example of using a Pipeline object, see Application Techniques.

Examples

This code in a Pipeline user object’s PipeMeter event report the number of rows that have been piped to the destination database:

```plaintext
ist_status.Text = String(This.RowsWritten) &
                  + " rows written to the destination database."
```

See also

PipeEnd
PipeStart
**PipeStart**

Description: Occurs when pipeline processing begins.

Event ID:

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_pipestart</td>
<td>Pipeline</td>
</tr>
</tbody>
</table>

Arguments: None

Return codes: Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

Usage: You can use the PipeStart event to check the status of pipeline processing.

The Start and Repair functions initiate pipeline processing.

For a complete example of using a Pipeline object, see *Application Techniques*.

Examples:

This code in a Pipeline user object’s PipeStart event reports pipeline status in a StaticText control:

```
  ist_status.Text = "Beginning Pipeline Execution ...
```

See also: PipeEnd, PipeMeter

**PrintFooter**

Description: Occurs when the footer of a page of the document in the RichTextEdit control is about to be printed.

Obsolete event

The PrintHeader and PrintFooter events are obsolete. They are no longer triggered under any circumstance. You must use the ShowHeadFoot function to edit headers and footers of pages in a rich text control at runtime.

Event ID:

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renprintfooter</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>
PrintHeader

Description
Occurs when the header of a page of the document in the RichTextEdit control is about to be printed.

Obsolete event
The PrintHeader and PrintFooter events are obsolete. They are no longer triggered under any circumstance. You must use the ShowHeadFooter function to edit headers and footers of pages in a rich text control at runtime.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renprintheader</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

ProgressIndex

Description
Occurs periodically during synchronization after updates to a synchronization progress bar.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>progress_idx</td>
<td>Long value representing the progress of the synchronization.</td>
</tr>
<tr>
<td>progress_max</td>
<td>Long value indicating the progress limit of the synchronization.</td>
</tr>
</tbody>
</table>

Return codes
None

Usage
Use this event to update a progress indicator such as a progress bar.

A progress_max value of 0 indicates the maximum value has not changed since the last time the event was fired.

See also
BeginLogScan
EndLogScan
CHAPTER 9  PowerScript Events

PropertyChanged

Description  Occurs after the OLE server changes the value of a property of the OLE object.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>OLE</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>propertyname</td>
<td>The name of the property whose value changed. If propertyname is an empty string, a more general change occurred, such as changes to more than one property.</td>
</tr>
</tbody>
</table>

Return codes  None (do not use a RETURN statement)

Usage  Property change notifications are not supported by all OLE servers. The PropertyRequestEdit and PropertyChanged events occur only when the server supports these notifications.

Property notifications are not sent when the object is being created or loaded. Otherwise, notifications are sent for all bindable properties, no matter how the property is being changed.

The PropertyChanged event occurs after the property’s value has changed. You can obtain the new value through the automation interface. The change can no longer be canceled. If you want to cancel a change, write a script for the PropertyRequestEdit event.

See also  DataChange  PropertyRequestEdit  Rename  ViewChange
PropertyRequestEdit

Description
Occurs when the OLE server is about to change the value of a property of the object in the OLE control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>OLE</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>propertyname</td>
<td>String by value (the name of the property whose value changed). If propertyname is an empty string, a more general change occurred, such as changes to more than one property.</td>
</tr>
</tbody>
</table>
| cancelchange | Boolean by reference; determines whether the change will be canceled. Values are:
  - FALSE – (Default) the change is allowed.
  - TRUE – the change is canceled. |

Return codes
None. Do not use a RETURN statement.

Usage
Property change notifications are not supported by all OLE servers. The PropertyRequestEdit and PropertyChanged events only occur when the server supports these notifications.

Property notifications are not sent when the object is being created or loaded. Otherwise, notifications are sent for all bindable properties, no matter how the property is being changed.

The PropertyRequestEdit event gives you a chance to access the property’s old value using the automation interface and save it. To cancel the change, set the cancelchange argument to true.

See also
DataChange
PropertyChanged
Rename
ViewChange
RButtonDown

The RButtonDown event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controls and windows, except RichTextEdit</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>RichTextEdit control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1

**For controls and windows, except RichTextEdit**

**Description**

For a window, occurs when the right mouse button is pressed in an unoccupied area of the window (any area with no visible, enabled object). The window event will occur if the cursor is over an invisible or disabled control.

For a control, occurs when the right mouse button is pressed on the control.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_rbuttondown</td>
<td>Windows and controls that can be placed on a window, except RichTextEdit</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| flags    | UnsignedLong by value (the modifier keys and mouse buttons that are pressed).
|          | Values are: |
|          | • 1 – Left mouse button |
|          | • 2 – Right mouse button |
|          | • 4 – Shift key |
|          | • 8 – Ctrl key |
|          | • 16 – Middle mouse button |

In the RButtonDown event, the right mouse button is always pressed, so 2 is always summed in the value of flags.

For an explanation of flags, see Syntax 2 of MouseMove on page 269.

<table>
<thead>
<tr>
<th>xpos</th>
<th>Integer by value (the distance of the pointer from the left edge of the window’s workspace in pixels).</th>
</tr>
</thead>
<tbody>
<tr>
<td>ypos</td>
<td>Integer by value (the distance of the pointer from the top of the window’s workspace in pixels).</td>
</tr>
</tbody>
</table>
**RButtonDown**

Return codes

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Examples

These statements in the RButtonDown script for the window display a pop-up 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 and therefore needs to be created. NewMenu is an instance of Menu4 (datatype Menu4):

```plaintext
Menu4 NewMenu
NewMenu = CREATE Menu4
NewMenu.m_language.PopMenu(xpos, ypos)
```

In a Multiple Document Interface (MDI) application, the arguments for PopMenu need to specify coordinates relative to the MDI frame:

```plaintext
NewMenu.m_language.PopMenu( &
   w_frame.PointerX(), w_frame.PointerY())
```

See also  Clicked

---

**Syntax 2**  For RichTextEdit controls

**Description**

Occurs when the user presses the right mouse button on the RichTextEdit control and the control’s PopMenu property is set to false.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renrbuttondown</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

**Arguments**

None

**Return codes**

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

**Usage**

If the control’s PopMenu property is true, the standard RichTextEdit pop-up menu is displayed instead, and the RButtonDown event does not occur.

You can use the RButtonDown event to implement your own pop-up menu.

See also  Clicked

RButtonDown
**RButtonUp**

**Description**
Occurs when the right mouse button is released.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renrbuttonup</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code choices (specify in a RETURN statement):
- 0  Continue processing
- 1  Prevent processing

**See also**
RButtonDown

---

**RecognitionResult**

**Description**
Occurs when an InkEdit control gets results from a call to the RecognizeText function.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_inkkerrecognition</td>
<td>InkEdit</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
None

**Examples**
This code in the RecognitionResult event allows the application to wait a few seconds while the Text property of the io_id InkEdit control is updated, then writes the recognized text to the string variable ls_inktext:

```powerscript
Sleep(3)
ls_inktext = ie_id.Text
```

**See also**
GetFocus
Stroke
**RemoteExec**

**Description**

Occurs when a DDE client application has sent a command.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_ddeexecute</td>
<td>Window</td>
</tr>
</tbody>
</table>

**Arguments**

None

**Return codes**

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

**See also**

RemoteRequest
RemoteSend

---

**RemoteHotLinkStart**

**Description**

Occurs when a DDE client application wants to start a hot link.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_ddeadvise</td>
<td>Window</td>
</tr>
</tbody>
</table>

**Arguments**

None

**Return codes**

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

**Examples**

When both the DDE client and server are PowerBuilder applications, this example in a script in the client application triggers the RemoteHotLinkStart event in the server application window:

```powershell
StartHotLink("mysle","pb_dde_server","mytest")
```

In the RemoteHotLinkStart event in the server application, set a boolean instance variable indicating that a hot link has been established:

```powershell
ib_hotlink = TRUE
```

**See also**

HotLinkAlarm
RemoteHotLinkStop
SetDataDDE
StartServerDDE
StopServerDDE
RemoteHotLinkStop

Description
Occurs when a DDE client application wants to end a hot link.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_ddeunadvise</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):
0  Continue processing

Examples
When both the DDE client and server are PowerBuilder applications, this example in a script in the client application triggers the RemoteHotLinkStop event in the server application window:

```powerbuilder
StopHotLink("mysle","pb_dde_server","mytest")
```

In the RemoteHotLinkStart event in the server application, set a boolean instance variable indicating that a hot link no longer exists:

```powerbuilder
ib_hotlink = FALSE
```

See also
HotLinkAlarm
RemoteHotLinkStart
SetDataDDE
StartServerDDE
StopServerDDE

RemoteRequest

Description
Occurs when a DDE client application requests data.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_dderequest</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):
0  Continue processing

See also
RemoteExec
RemoteSend
RemoteSend

Description
Occurs when a DDE client application has sent data.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_ddepoke</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):
- 0  Continue processing

See also
RemoteExec
RemoteRequest

Rename

Description
Occurs when the server application notifies the control that the object has been renamed.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_omnrename</td>
<td>OLE</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code: Ignored

Usage
If you want to retrieve the ObjectData blob value of an OLE control during the processing of this event, you must post a user event back to the control or you will generate a runtime error.

See also
DataChange
PropertyRequestEdit
PropertyChanged
ViewChange
Resize

Description
Occurs when the user or a script opens or resizes the client area of a window or DataWindow control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_dwnresize</td>
<td>DataWindow</td>
</tr>
<tr>
<td>pbm_size</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizetype</td>
<td>UnsignedLong by value. The values are:</td>
</tr>
<tr>
<td></td>
<td>• 0 – (SIZE_RESTORED) The window or DataWindow has been resized, but it was not minimized or maximized. The user might have dragged the borders or a script might have called the Resize function.</td>
</tr>
<tr>
<td></td>
<td>• 1 – (SIZE_MINIMIZED) The window or DataWindow has been minimized.</td>
</tr>
<tr>
<td></td>
<td>• 2 – (SIZE_MAXIMIZED) The window or DataWindow has been maximized.</td>
</tr>
<tr>
<td></td>
<td>• 3 – (SIZE_MAXSHOW) This window is a pop-up window and some other window in the application has been restored to its former size (does not apply to DataWindow controls).</td>
</tr>
<tr>
<td></td>
<td>• 4 – (SIZE_MAXHIDE) This window is a pop-up window and some other window in the application has been maximized (does not apply to DataWindow controls).</td>
</tr>
<tr>
<td>newwidth</td>
<td>Integer by value (the width of the client area of a window or DataWindow control in PowerBuilder units).</td>
</tr>
<tr>
<td>newheight</td>
<td>Integer by value (the height of the client area of a window or DataWindow control in PowerBuilder units).</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):

0    Continue processing
RightClicked

The RightClicked event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView and Tab control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Tab control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1

**For ListView and Tab controls**

**Description**

Occurs when the user clicks the right mouse button on the ListView control or the tab portion of the Tab control.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnrclicked</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_tcnrclicked</td>
<td>Tab</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value (the index of the item or tab the user clicked)</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Usage**

When the user clicks in the display area of the Tab control, the tab page user object gets an RButtonDown event rather than a RightClicked event for the Tab control.

**Examples**

This example for the RightClicked event of a ListView control displays a pop-up menu when the user clicks the right mouse button:

```powerbuilder
// Declare a menu variable of type m_main
m_main m_lv_popmenu
// Create an instance of the menu variable
m_lv_popmenu = CREATE m_main
// Display menu at pointerposition
m_lv_popmenu.m_entry.PopMenu(Parent.PointerX(), &Parent.PointerY())
```

**See also**

Clicked
RightDoubleClicked
Syntax 2 For TreeView controls

Description
Occurs when the user clicks the right mouse button on the TreeView control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnrclicked</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by value (the handle of the item the user clicked)</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

Examples
This example for the RightClicked event of a TreeView control displays a pop-up menu when the user clicks the right mouse button:

```PowerScript
// Declare a menu variable of type m_main
m_main m_tv_popmenu

// Create an instance of the menu variable
m_tv_popmenu = CREATE m_main

// Display menu at pointer position
m_tv_popmenu.m_entry.PopMenu(Parent.PointerX(), &Parent.PointerY())
```

See also
Clicked
RightDoubleClicked

RightDoubleClicked

The RightDoubleClicked event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView and Tab control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>
Syntax 1  
**For ListView and Tab controls**

**Description**
Occurs when the user double-clicks the right mouse button on the ListView control or the tab portion of the Tab control.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnrdoubleclicked</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_tcnrdoubleclicked</td>
<td>Tab</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value (the index of the item or tab the user double-clicked)</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing

**Examples**
This example deletes an item from the ListView when the user right-double-clicks on it and then rearranges the items:

```powerbuilder
integer li_rtn

// Delete the item
li_rtn = This.DeleteItem(index)

IF li_rtn = 1 THEN
   This.Arrange()
ELSE
   MessageBox("Error", Deletion failed!"
END IF
```

**See also**
DoubleClicked
RightClicked

Syntax 2  
**For TreeView controls**

**Description**
Occurs when the user double-clicks the right mouse button on the TreeView control.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnrdoubleclicked</td>
<td>TreeView</td>
</tr>
</tbody>
</table>
CHAPTER 9  PowerScript Events

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Long by value (the handle of the item the user double-clicked)</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>

Examples

This example toggles between displaying and hiding TreeView lines when the user right-double-clicks on the control:

```powerpax
IF This.HasLines = FALSE THEN
    This.HasLines = TRUE
    This.LinesAtRoot = TRUE
ELSE
    This.HasLines = FALSE
    This.LinesAtRoot = FALSE
END IF
```

See also

DoubleClicked
RightClicked

Save

Description

Occurs when the server application notifies the control that the data has been saved.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_omnsave</td>
<td>OLE</td>
</tr>
</tbody>
</table>

Arguments

None

Return codes

Long. Return code: Ignored

Usage

If you want to retrieve the ObjectData blob value of an OLE control during the processing of this event, you must post a user event back to the control or you generate a runtime error.

Examples

In this example, a table in a database tracks changes of OLE objects; when the user saves an Excel spreadsheet in an OLE control, this code puts the current date in a DataWindow so that the database table can be updated:

```powerpax
long ll_row
// Find the row with information for the Excel file
```
ll_row = dw_1.Find("file_name = 'expenses.xls'", & 1, 999)

IF ll_row > 0 THEN
  // Make the found row current
  dw_1.SetRow(ll_row)

  // Put today's date in the last_updated column
  dw_1.Object.last_updated[ll_row] = Today( )

  // Update and refresh the DataWindow
  dw_1.Update( )
dw_1.Retrieve( )
ELSE
  MessageBox("Find", "No row found")
END IF

See also
Close
SaveObject

SaveObject

Description
Occurs when the server application saves the object in the control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_omnsaveobject</td>
<td>OLE</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code: Ignored

Usage
Using the SaveObject event is the preferred technique for retrieving the ObjectData blob value of an OLE control when the server saves the data in the embedded object. Unlike the Save and Close events, the SaveObject event does not require you to post a user event back to the control to prevent the generation of a runtime error.
Because of differences in the behavior of individual servers, this event is not triggered consistently across all server applications. Using Microsoft Word or Excel, the SaveObject event is triggered when the DisplayType property of the control is set to DisplayAsActiveXDocument! or DisplayAsIcon!, but not when it is set to DisplayAsContent!. For other applications, such as Paint Shop Pro, the event is triggered when the display type is DisplayAsContent! but not when it is DisplayAsActiveXDocument!.

Because some servers might also fire the PowerBuilder Save event and the relative timing of the two events cannot be guaranteed, your program should handle only the SaveObject event.

**Examples**

In this example, when the user or the server application saves a Word document in an OLE control, the data is saved as a blob in a file. The file can then be opened as a Word document:

```powerbuilder
blob  l_myobjectdata
l_myobjectdata = this.objectdata
integer  l_file

l_file = FileOpen("c:\myfile.doc", StreamMode!, Write!)
FileWrite( l_file, l_myobjectdata )
FileClose( l_file )
```

**See also**

Close
Save

---

**Selected**

**Description**

Occurs when the user highlights an item on the menu using the arrow keys or the mouse, without choosing it to be executed.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Menu</td>
</tr>
</tbody>
</table>

**Arguments**

None

**Return codes**

None. (Do not use a RETURN statement.)

**Usage**

You can use the Selected event to display MicroHelp for the menu item. One way to store the Help text is in the menu item’s Tag property.
SelectionChanged

Examples

This example uses the tag value of the current menu item to display Help text. The function `wf_SetMenuHelp` takes the text passed (the tag) and assigns it to a MultiLineEdit control. A Timer function and the Timer event are used to clear the Help text.

This code in the Selected event calls the function that sets the text:

```plaintext
w_test.wf_SetMenuHelp(This.Tag)
```

This code for the `wf_SetMenuHelp` function sets the text in the MultiLineEdit `mle_menuhelp`; its argument is called `menuhelpstring`:

```plaintext
mle_menuhelp.Text = menuhelpstring
    Timer(4)
```

This code in the Timer event clears the Help text and stops the timer:

```plaintext
w_test.wf_SetMenuHelp(""
    Timer(0)
```

See also

Clicked

SelectionChanged

The SelectionChanged event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>DropDownListBox, DropDownPictureListBox, ListBox, PictureListBox controls</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Tab control</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

Syntax 1

For Listboxes

Description

Occurs when an item is selected in the control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_chnselchange</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_lbnselchange</td>
<td>ListBox, PictureListBox</td>
</tr>
</tbody>
</table>
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Integer by value (the index of the item that has become selected)</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage

For DropDownListBoxes, the SelectionChanged event applies to selections in the drop-down portion of the control, not the edit box.

The SelectionChanged event occurs when the user clicks on any item in the list, even if it is the currently selected item. When the user makes a selection using the mouse, the Clicked (and if applicable the DoubleClicked event) occurs after the SelectionChanged event.

Examples

This example is for the lb_value ListBox in the window w_graph_sheet_with_list in the PowerBuilder Examples application. When the user chooses values, they are graphed as series in the graph gr_1. The MultiSelect property for the ListBox is set to true, so index has no effect. The script checks all the items to see if they are selected:

```powerbuilder
integer itemcount, i, r
string ls_colname

gr_1.SetRedraw(FALSE)

// Clear out categories, series and data from graph
gr_1.Reset(All!)

// Loop through all selected values and
// create as many series as the user specified
FOR i = 1 to lb_value.TotalItems()
    IF lb_value.State(i) = 1 THEN
        ls_colname = lb_value.Text(i)

        // Call window function to set up the graph
        wf_set_a_series(ls_colname, ls_colname, &
                        lb_category.text(1))
    END IF
NEXT

gr_1.SetRedraw(TRUE)
```

See also

Clicked
**SelectionChanged**

### Syntax 2
For Tab controls

**Description**
Occurs when a tab is selected.

**Event ID**
- Event ID: pbm_tcnse1changed
  - Objects: Tab

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldindex</td>
<td>Integer by value (the index of the tab that was previously selected)</td>
</tr>
<tr>
<td>newindex</td>
<td>Integer by value (the index of the tab that has become selected)</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specify in a RETURN statement):
- 0 Continue processing

**Usage**
The SelectionChanged event occurs when the Tab control is created and the initial selection is set.

**See also**
Clicked
SelectionChanging

### Syntax 3
For TreeView controls

**Description**
Occurs when the item is selected in a TreeView control.

**Event ID**
- Event ID: pbm_tvse1changed
  - Objects: TreeView

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldhandle</td>
<td>Long by value (the handle of the previously selected item)</td>
</tr>
<tr>
<td>newhandle</td>
<td>Long by value (the handle of the currently selected item)</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code choices (specify in a RETURN statement):
- 0 Continue processing

**Usage**
The SelectionChanged event occurs after the SelectionChanging event.

**Examples**
This example tracks items in the SelectionChanged event:

```
TreeViewItem1_tvinew, l_tviold
```
// get the treeview item that was the old selection
This.GetItem(oldhandle, l_tviold)

// get the treeview item that is currently selected
This.GetItem(newhandle, l_tvinew)

// Display the labels for the two items in sle_get
sle_get.Text = "Selection changed from " &
+ String(l_tviold.Label) + " to " &
+ String(l_tvinew.Label)

See also
Clicked
SelectionChanging

SelectionChanging

The SelectionChanging event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tab control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

For Tab controls

Description
Occurs when another tab is about to be selected.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tcnselechanging</td>
<td>Tab</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldindex</td>
<td>Integer by value (the index of the currently selected tab)</td>
</tr>
<tr>
<td>newindex</td>
<td>Integer by value (the index of the tab that is about to be selected)</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

0  Allow the selection to change
1  Prevent the selection from changing
SelectionChanging

Usage
Use the SelectionChanging event to prevent the selection from changing or to do processing for the newly selected tab page before it becomes visible. If CreateOnDemand is true and this is the first time the tab page is selected, the controls on the page do not exist yet, and you cannot refer to them in the event script.

Examples
When the user selects a tab, this code sizes the DataWindow control on the tab page to match the size of another DataWindow control. The resizing happens before the tab page becomes visible. This example is from tab_uo in the w_phone_dir window in the PowerBuilder Examples:

```pascal
u_tab_dirluo_Tab
luo_Tab = This.Control[newindex]
luo_Tab.dw_dir.Height = dw_list.Height
luo_Tab.dw_dir.Width = dw_list.Width
```

See also
Clicked
SelectionChanged

Syntax 2
For TreeView controls
Description
Occurs when the selection is about to change in the TreeView control.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnselchanging</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldhandle</td>
<td>Long by value (the handle of the currently selected item)</td>
</tr>
<tr>
<td>newhandle</td>
<td>Long by value (the handle of the item that is about to be selected)</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Allow the selection to change</td>
</tr>
<tr>
<td>1</td>
<td>Prevent the selection from changing</td>
</tr>
</tbody>
</table>

Usage
The SelectionChanging event occurs before the SelectionChanged event.

Examples
This example displays the status of changing TreeView items in a SingleLineEdit:

```pascal
TreeViewItem l_tvinew, l_tviold

// Get TreeViewItem that was the old selection
```
CHAPTER 9  PowerScript Events

This.GetItem(oldhandle, l_tviold)

    // Get TreeViewItem that is currently selected
This.GetItem(newhandle, l_tvinew)

    // Display the labels for the two items in display
sle_status.Text = "Selection changed from " &
    + String(l_tviold.Label) + " to " &
    + String(l_tvinew.Label)

See also

    Clicked
    SelectionChanged

Show

Description

Occurs just before the window is displayed.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_showwindow</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>show</td>
<td>Boolean by value (whether the window is being shown). The value is always true.</td>
</tr>
<tr>
<td>status</td>
<td>Long by value (the status of the window). Values are:</td>
</tr>
<tr>
<td></td>
<td>• 0 – The current window is the only one affected.</td>
</tr>
<tr>
<td></td>
<td>• 1 – The window’s parent is also being minimized or a pop-up window is being hidden.</td>
</tr>
<tr>
<td></td>
<td>• 3 – The window’s parent is also being displayed or maximized or a pop-up window is being shown.</td>
</tr>
</tbody>
</table>

Return codes

Long. Return code choices (specify in a RETURN statement):

    0  Continue processing

Usage

The Show event occurs when the window is opened.

See also

    Activate
    Hide
    Open
### Sort

The Sort event has different arguments for different objects:

<table>
<thead>
<tr>
<th>Object</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1: For ListView controls

**Description**

Occurs for each comparison when the ListView is being sorted.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lvnsort</td>
<td>ListView</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index1</td>
<td>Integer by value (the index of one item being compared during a sorting operation)</td>
</tr>
<tr>
<td>index2</td>
<td>Integer by value (the index of the second item being compared)</td>
</tr>
<tr>
<td>column</td>
<td>Integer by value (the number of the column containing the items being sorted)</td>
</tr>
</tbody>
</table>

**Return codes**

Long. Return code choices (specify in a RETURN statement):

-1  \( \text{index1} \) is less than \( \text{index2} \)  
0  \( \text{index1} \) is equal to \( \text{index2} \)  
1  \( \text{index1} \) is greater than \( \text{index2} \)  

**Usage**

The Sort event allows you to fine-tune the sort order of the items being sorted. You can examine the properties of each item and tell the Sort function how to sort them by selecting one of the return codes.

You typically use the Sort event when you want to sort ListView items based on multiple criteria such as a PictureIndex and Label.

The Sort event occurs if you call the Sort event, or when you call the Sort function using the UserDefinedSort! argument.

**Examples**

This example sorts ListView items according to PictureIndex and Label sorting by PictureIndex first, and then by label:

```powershell
ListViewItem lvi, lvi2
```
This.GetItem(index1, lvi)
This.GetItem(index2, lvi2)

IF lvi.PictureIndex > lvi2.PictureIndex THEN
    RETURN 1
ELSEIF lvi.PictureIndex < lvi2.PictureIndex THEN
    RETURN -1
ELSEIF lvi.label > lvi2.label THEN
    RETURN 1
ELSEIF lvi.label < lvi2.label THEN
    RETURN -1
ELSE
    RETURN 0
END IF

Syntax 2 For TreeView controls
Description
Occurs for each comparison when the TreeView is being sorted.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnsort</td>
<td>TreeView</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle1</td>
<td>Long by value (the handle of one item being compared during a sorting operation)</td>
</tr>
<tr>
<td>handle2</td>
<td>Long by value (the handle of the second item being compared)</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code choices (specify in a RETURN statement):
-1 handle1 is less than handle2
0 handle1 is equal to handle2
1 handle1 is greater than handle2

Usage
The Sort event allows you to fine-tune the sort order of the items being sorted. You can examine the properties of each item and tell the Sort function how to sort them by selecting one of the return codes.

You typically use the Sort event when you want to sort TreeView items based on multiple criteria such as a PictureIndex and Label.

The Sort event occurs if you call the Sort event, or when you call the Sort function using the UserDefinedSort! argument.
Examples
This example sorts TreeView items according to PictureIndex and Label
sorting by PictureIndex first, then by label:

```
TreeViewItem tvi, tvi2

This.GetItem(handle1, tvi)
This.GetItem(handle2, tvi2)

IF tvi.PictureIndex > tvi2.PictureIndex THEN
    RETURN 1
ELSEIF tvi.PictureIndex < tvi2.PictureIndex THEN
    RETURN -1
ELSEIF tvi.Label > tvi2.Label THEN
    RETURN 1
ELSEIF tvi.Label < tvi2.Label THEN
    RETURN -1
ELSE
    RETURN 0
END IF
```

SQLPreview
Description
Occurs immediately before a SQL statement is submitted to the DBMS.

Event ID
<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Transaction objects</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlfunc</td>
<td>The SQLFunction system enumeration passed by value. This enumeration indicates the function that initiated database activity. Values for this enumeration are: SQLDBInsert!, SQLDBUpdate!, SQLDBDelete!, SQLDBSelect!, SQLDBProcedure!, SQLDBRPC!, and SQLDBOthers!.</td>
</tr>
<tr>
<td>sqlsyntax</td>
<td>String by value. The full text of the SQL statement.</td>
</tr>
</tbody>
</table>

Return codes
Long. Set the return code to affect the outcome of the event:

```
0  Continue processing
1  Stop processing
```

Usage
Use with embedded SQL  This event is triggered before SQL statements are passed to the DBMS.
Use with DataWindow/DataStore  When using this event of the Transaction object with a DataWindow that also defines a SQLPreview event, the DataWindow’s event is fired first. If the return value of DataWindow SQLPreview event is 0 (continue processing), the Transaction object’s SQLPreview event is triggered next.

Examples

In this embedded SQL cursor example, the SQLPreview event is invoked just before the OPEN Emp_curs; statement is executed:

    DECLARE Emp_cur CURSOR FOR SELECT employee.emp_number, employee.emp_name FROM employee WHERE employee.emp_salary > :Sal_var ;
    OPEN Emp_curs ;

In this dynamic SQL example, the SQLPreview event is invoked just before the EXECUTE DYNAMIC my_proc DESCRIPTOR SQLDA statement is executed:

    PREPARE SQLSA FROM  "execute @rc = myproc @parm1=?, @parm2=? OUTPUT ";
    DESCRIBE SQLSA INTO SQLDA ;
    DECLARE my_proc DYNAMIC PROCEDURE FOR SQLSA ;
    SetDynamicParm(SQLDA, 1, var1)
    SetDynamicParm(SQLDA, 2, var2)
    EXECUTE DYNAMIC my_proc DESCRIPTOR SQLDA ;

See also DBError
SQLPreview in the DataWindow Reference

Start

Description
Occurs when an animation has started playing.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_animatestart</td>
<td>Animation</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>

See also Stop
Stop

Description
Occurs when an animation has stopped playing.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_animatestop</td>
<td>Animation</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>

See also
Timer

Stroke

Description
Occurs when the user draws a new stroke.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_inkestroke, pbm_inkpstroke</td>
<td>InkEdit, InkPicture</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
Boolean. Return true to erase the stroke and false otherwise.

Usage
If the InkMode property of an InkEdit control is set to InkDisabled!, or the InkCollectionMode property of an InkPicture control is set to GestureOnly!, the Stroke event is not triggered.

See also
Gesture
RecognitionResult
SyncPreview

Description
Returns generated dbmlsync command arguments immediately prior to launching the synchronization process.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command_args</td>
<td>String passed by reference that includes dbmlsync command arguments for launching the synchronization process.</td>
</tr>
</tbody>
</table>

Return codes
None

Usage
Use this event to receive and display dbmlsync command line arguments.

The event is called by the Synchronize function just before it launches dbmlsync. The generated command arguments for dbmlsync are passed by reference in the command_args string. You can change the command_args string with PowerScript code or with the debugger. If command_args are changed, the Synchronize function will use the new argument string.

See also
BeginDownload
BeginSync

SystemError

Description
Occurs when a serious runtime error occurs (such as trying to open a nonexistent window) if the error is not handled in a try-catch block.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Application</td>
</tr>
</tbody>
</table>

Arguments
None

Return codes
None. (Do not use a RETURN statement.)

Usage
If there is no script for the SystemError event, PowerBuilder displays a messagebox with the PowerBuilder error number and error message text. For information about error messages, see the PowerBuilder Users Guide.
If you comment out the entire script in the SystemError event, a Windows Forms application behaves as if there is no script for the event and the message box displays, but in a standard PowerBuilder application the event is still triggered and the message box does not display.

For errors involving external objects and DataWindows, you can handle the error in the ExternalException or Error events and prevent the SystemError event from occurring. The ExternalException and Error events are maintained for backward compatibility.

You can prevent the SystemError event from occurring by handling errors in try-catch blocks. Well-designed exception-handling code gives application users a better chance to recover from error conditions and run the application without interruption. For information about exception handling, see Application Techniques.

When a SystemError event occurs, your current script terminates and your system might become unstable. It is generally not a good idea to continue running the application, but you can use the SystemError event script to clean up and disconnect from the DBMS before closing the application.

Examples

This statement in the SystemError event halts the application immediately:

```
HALT CLOSE
```

See also

Error
ExternalException
TRY...CATCH...FINALLY...END TRY

**SystemKey**

*Description*

Occurs when the insertion point is not in a line edit, and the user presses the Alt key (alone or with another key).

*Event ID*

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_syskeydown</td>
<td>Window</td>
</tr>
</tbody>
</table>

*Arguments*

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>KeyCode by value. A value of the KeyCode enumerated datatype indicating the key that was pressed, for example, KeyA! or KeyF1!.</td>
</tr>
</tbody>
</table>
CHAPTER 9  PowerScript Events

Return codes
Long. Return code choices (specify in a RETURN statement):

0  Continue processing

Usage
Pressing the Ctrl key prevents the SystemKey event from firing when the Alt key is pressed.

Examples
This example displays the name of the key that was pressed with the Alt key:

string ls_key

CHOOSE CASE key

CASE KeyF1!
    ls_key = "F1"
CASE KeyA!
    ls_key = "A"
CASE KeyF2!
    ls_key = "F2"
END CHOOSE

This example causes a beep if the user presses Alt+Shift+F1.

IF keyflags = 1 THEN
    IF key = KeyF1 THEN
        Beep(1)
    END IF
END IF

See also
Key

Timer

Description
Occurs when a specified number of seconds elapses after the Start or Timer function has been called.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_timer</td>
<td>Timing or Window</td>
</tr>
</tbody>
</table>

Arguments
None
### Return codes

Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>

### Examples

These examples show how to use a timing object’s Timer event and a window’s Timer event.

**Using a timing object**  
This example uses a timing object to refresh a list of customers retrieved from a database at specified intervals. The main window of the application, `w_main`, contains a DataWindow control displaying a list of customers and two buttons, Start Timer and Retrieve. The window’s Open event connects to the database:

```sql
CONNECT using SQLCA;

IF sqlca.sqlcode <> 0 THEN
    MessageBox("Database Connection", & sqlca.sqlerrtext)
END IF
```

The following code in the clicked event of the Start Timer button creates an instance of a timing object, `nvo_timer`, and opens a response window to obtain a timing interval. Then, it starts the timer with the specified interval:

```powershell
MyTimer = CREATE nvo_timer
open(w_interval)
MyTimer.Start(d_interval)

MessageBox("Timer", "Timer Started. Interval is " & + string(MyTimer.interval) + " seconds")
```

In the timing object’s Constructor event, the following code creates an instance of a datastore:

```powershell
ds_datastore = CREATE datastore
```

The timing object’s Timer event calls an object-level function called `refresh_custlist` that refreshes the datastore. This is the code for `refresh_custlist`:

```c
long ll_rowcount

d_datastore.dataobject = "d_customers"
d_datastore.SetTransObject (SQLCA)
ll_rowcount = ds_datastore.Retrieve()

RETURN ll_rowcount
```

The Retrieve button on `w_main` simply shares the data from the DataStore with the DataWindow control:

```powershell
ds_datastore.ShareData(dw_1)```
Using a window object  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, this code displays the time initially and starts the timer:

```powerscript
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:

```powerscript
st_time.Text = String(Now(), "hh:mm")
```

See also  Start  Timer

**ToolbarMoved**

**Description**  Occurs in an MDI frame window when the user moves any FrameBar or SheetBar.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tbnmoved</td>
<td>Window</td>
</tr>
</tbody>
</table>

**Arguments**  None

**Return codes**  Long. Return code choices (specify in a RETURN statement):

<table>
<thead>
<tr>
<th>Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Continue processing</td>
</tr>
</tbody>
</table>

**Usage**  The event is not triggered for sheet windows.

To get information about the toolbars’ positions, call the GetToolbar and GetToolbarPos functions.

This event occurs when you change a toolbar’s position with SetToolbarPos.
## UploadAck

### Description
Occurs on completion of upload processing.

### Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uploadack_status</td>
<td>Long indicating the status returned by MobiLink to the remote after the upload stream is processed. Values are:</td>
</tr>
<tr>
<td></td>
<td>• 0 for StatCommitted</td>
</tr>
<tr>
<td></td>
<td>• 1 for StatFailed</td>
</tr>
<tr>
<td></td>
<td>• 248 for StatRetry</td>
</tr>
<tr>
<td></td>
<td>See the Usage note for the meaning of these values.</td>
</tr>
</tbody>
</table>

### Return codes
None

### Usage
Use this event to add custom actions after dbmlsync has received acknowledgement of the upload stream from the MobiLink synchronization server.

Values for the uploadack_status argument can be:

- **StatCommitted** Indicates that the upload stream was received by the MobiLink synchronization server and committed.

- **StatRetry** Indicates that the MobiLink synchronization server and the remote database had different values for the log offset from which the upload stream should start. The upload stream was not committed by the MobiLink synchronization server. The component attempts to send another upload stream starting from the MobiLink synchronization server's log offset.

- **StatFailed** Indicates that the MobiLink synchronization server did not commit the upload stream.

### See also
- BeginUpload
- ConnectMobiLink
- EndUpload
- WaitForUploadAck
UserString

Description
Occurs when the user has edited the contents of the control and the control has lost focus. The AllowEdit property must be set to true.

Event ID

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_dtpuserstring</td>
<td>DatePicker</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag</td>
<td>Unsigned long by reference. The value of flag is 0 by default and should not be changed.</td>
</tr>
<tr>
<td>userstr</td>
<td>String entered in the control by the user.</td>
</tr>
<tr>
<td>dtm</td>
<td>A DateTime value by reference to which the validated date should be assigned.</td>
</tr>
</tbody>
</table>

Return codes
Long. Return code: Ignored.

Usage
When a user tabs into a DatePicker control, it is in normal editing mode and one part of the date (year, month, or day) can be edited. If the AllowEdit property is set to true, the user can press F2 or click in the control to select all the text in the control for editing. When the control loses focus, the control returns to normal editing mode and the UserString event is fired, allowing you to test whether the text in the control is a valid date. The UserString event fires whether or not the text was modified.

The text entered in the control must be in a format that can be converted into a valid DateTime variable. If the string entered by the user can be converted to a valid DateTime value, you can assign the parsed DateTime value to the dtm argument to change the Value property of the control.

The ValueChanged event is fired after the UserString event if the value changed.

Examples
This code in the UserString event script tests whether the string entered by the user is a valid date. If it is valid, the code converts the date to a DateTime so that it can be assigned to the DatePicker’s Value property. Otherwise it displays an error message to the user:

```powerShell
IF IsDate(userstr) THEN
  dtm = DateTime(Date(userstr))
ELSE
  MessageBox("Invalid date", userstr)
END IF
```
**ValueChanged**

**Description**
Occurs when the Value property in a DatePicker control changes.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_dtpvaluechanged</td>
<td>DatePicker</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag</td>
<td>Unsigned long that defaults to 0 and can be ignored</td>
</tr>
<tr>
<td>dtm</td>
<td>The new DateTime value</td>
</tr>
</tbody>
</table>

**Return codes**
Long. Return code: Ignored.

**Usage**
When a user selects a date from the drop-down calendar or changes the date using the up-down control, the DateTime value selected is passed to the ValueChanged event.

**Examples**
This code in the ValueChanged event script displays a confirmation message to the user:

```powershell
MessageBox("Start date", "You selected " + &
        String(dtm, "mmm dd, yyyy") + ".n" + 
        "If this is incorrect, please select again.")
```

---

**ViewChange**

**Description**
Occurs when the server application notifies the control that the view shown to the user has changed.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_omnviewchange</td>
<td>OLE</td>
</tr>
</tbody>
</table>

**Arguments**
None

**Return codes**
Long. Return code: Ignored

**Usage**
If you want to retrieve the ObjectData blob value of an OLE control during the processing of this event, you must post a user event back to the control or you will generate a runtime error.

**See also**
DataChange
PropertyRequestEdit
PropertyChanged
Rename
### WaitForUploadAck

**Description**  
Occurs when the synchronization process starts a new waiting period for upload acknowledgement.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

**Arguments**  
None

**Return codes**  
None

**Usage**  
Use this event to add custom actions when the component is waiting for upload acknowledgement from the MobiLink synchronization server.

**See also**  
BeginUpload  
EndUpload

### WarningMessage

**Description**  
Occurs on display of a warning message.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>MLSynchronization, MLSync</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>warnmsg</td>
<td>Read-only string containing the text of the warning message returned from the synchronization server.</td>
</tr>
</tbody>
</table>

**Usage**  
Use this event to receive warning information logged by dbmilsync.

The following events can be triggered when different types of messages are sent by the synchronization server: DisplayMessage, ErrorMessage, FileMessage, and WarningMessage.

**See also**  
DisplayMessage  
ErrorMessage  
FileMessage
This chapter provides syntax, descriptions, and examples for PowerScript functions.

The functions are listed alphabetically.

For information about functions that apply to DataWindows or DataStores, see also the \textit{DataWindow Reference}. Methods that apply to DataWindows, but not to other PowerBuilder controls, are listed only in the \textit{DataWindow Reference}. 

See also
Abs

Description
Calculates the absolute value of a number.

Syntax
Abs ( n )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The number for which you want the absolute value</td>
</tr>
</tbody>
</table>

Return value
The datatype of n. Returns the absolute value of n. If n is null, Abs returns null.

Examples
All these statements set num to 4:

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

This statement returns 4.2:

```pascal
Abs (-4.2)
```

See also
Abs method for DataWindows in the DataWindow Reference or online Help

ACos

Description
Calculates the arccosine of an angle.

Syntax
ACos ( n )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The ratio of the lengths of two sides of a triangle for which you want a corresponding angle (in radians). The ratio must be a value between -1 and 1.</td>
</tr>
</tbody>
</table>

Return value
Double. Returns the arccosine of n.

Examples
This statement returns 0:

```pascal
ACos (1)
```

This statement returns 3.141593 (rounded to six places):

```pascal
ACos (-1)
```
This statement returns 1.000000 (rounded to six places):

\[ \text{ACos}(0.540302) \]

This code in the Clicked event of a button catches a runtime error that occurs when an arccosine is taken for a user-entered value—passed in a variable—that is outside of the permitted range:

```pascal
Double ld_num
ld_num = Double (sle_1.text)

TRY
sle_2.text = string (acos (ld_num))
CATCH (runtimeerror er)
       MessageBox("Runtime Error", er.getmessage())
END TRY
```

See also

- Cos
- ASin
- ATan
- ACos method for DataWindows in the DataWindow Reference or online Help

## Activate

**Description**

Activates the object in an OLE container, allowing the user to work with the object using the server’s commands.

**Applies to**

OLE controls and OLE DWOObjects (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 DWOObject within a DataWindow control that contains the object you want to activate.

The fully qualified name for a DWOObject has this syntax:

```
dwcontrol.Object.dwobjectname
```
**Activate**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activationtype (optional)</td>
<td>A value of the enumerated datatype omActivateType specifying where the user will work with the OLE object. Values are:</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• OffSite! – The object is activated in the server application, which gives the user access to more of the server application’s functionality.</td>
</tr>
<tr>
<td></td>
<td>For the OLE control, activationtype is required.</td>
</tr>
</tbody>
</table>

**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 cannot 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:

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

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

```powershell
def integer result  
result = dw_1.Object.ole_graph.Activate(OffSite!)
```

**See also**

DoVerb  
OLEActivate in the DataWindow Reference or the online Help  
SelectObject
AddCategory

Description: Adds a new category to the category axis of a graph. AddCategory is for a category axis whose datatype 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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph to which you want to add a category.</td>
</tr>
<tr>
<td>categoryname</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 datatype 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. Category names are unique if they have different capitalization.

To add categories when the axis datatype 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, but it requires an additional argument to do so.

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

\[ \text{listviewname}\text{.AddColumn}( \text{label}, \text{alignment}, \text{width} ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control to which you want to add a column.</td>
</tr>
<tr>
<td>label</td>
<td>A string whose value is the name of the column you are adding.</td>
</tr>
</tbody>
</table>
| alignment    | A value of the enumerated datatype 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, in PowerBuilder units. |

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.

Use SetItem and SetColumn to change the values for existing items. To add new items, use AddItem. To create columns for the report view of a ListView control, use AddColumn.

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

```powerbuilder
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
AddItem  
DeleteColumn  
InsertColumn

330

PowerBuilder
AddData

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

<table>
<thead>
<tr>
<th>To add data to</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any graph type except scatter</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Scatter graphs</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

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**
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to add data to a series. The graph’s type should not be scatter.</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number that identifies the series to which you want to add data.</td>
</tr>
<tr>
<td>datavalue</td>
<td>The value of the data you want to add.</td>
</tr>
<tr>
<td>categoryvalue (optional)</td>
<td>The category for this data value on the category axis. The datatype of the categoryvalue should match the datatype of the category axis. In most cases you should include categoryvalue. Otherwise, an uncategorized value will be added to the series.</td>
</tr>
</tbody>
</table>

**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 datatype 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 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:

```powershell
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:

```powershell
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
CHAPTER 10  PowerScript Functions

Syntax 2  For scatter graphs
Description Adds a data point to a series in a scatter graph.
Syntax

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the scatter graph in which you want to add data to a series. The graph’s type should be scatter.</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number that identifies the series to which you want to add data.</td>
</tr>
<tr>
<td>xvalue</td>
<td>The x value of the data point you want to add.</td>
</tr>
<tr>
<td>yvalue</td>
<td>The y value of the data point you want to add.</td>
</tr>
</tbody>
</table>

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:

```powershell
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.

<table>
<thead>
<tr>
<th>To add an item to</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ListBox or DropDownListBox control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>A PictureListBox or DropDownPictureListBox control</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>A ListView control when you only need to specify the item name and picture index</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>A ListView control when you need to specify all the properties for the item</td>
<td>Syntax 4</td>
</tr>
</tbody>
</table>
**Syntax 1**  
For ListBox and DropDownListBox controls

**Description**  
Adds a new item to the list of values in a list box.

**Applies to**  
ListBox and DropDownListBox controls

**Syntax**  

```powershell
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 at runtime.

**Adding many items to a list with a horizontal scroll bar**  
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.

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

```powershell
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).
Syntax 2  
For PictureListBox and DropDownPictureListBox controls

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

Applies to
PictureListBox and DropDownPictureListBox controls

Syntax
`listboxname.AddItem(item, pictureindex)`

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

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 do not specify a picture index, the newly added item will not have a picture.

If you specify a picture index that does not 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 additional notes about items in list boxes, see Syntax 1.

Examples
This example adds the item Cardinal to the PictureListBox `plb_birds`:

```power_script
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)
```
AddItem

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.

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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control to which you are adding a picture or item</td>
</tr>
<tr>
<td>label</td>
<td>The name of the item you are adding</td>
</tr>
<tr>
<td>pictureindex</td>
<td>The index of the picture you want to associate with the newly added item</td>
</tr>
</tbody>
</table>

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.

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

```
1v_1.AddItem("Sanyo", 1)  
1v_1.AddItem("Onkyo", 1)  
1v_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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the List View control to which you are adding a picture or item</td>
</tr>
<tr>
<td>item</td>
<td>The ListViewItem variable containing properties of the item you are adding</td>
</tr>
</tbody>
</table>

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.

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.AddLargePicture ( picturename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control to which you are adding a bitmap, icon, or cursor</td>
</tr>
<tr>
<td>picturename</td>
<td>The name of the bitmap, icon, or cursor you are adding to the large image list</td>
</tr>
</tbody>
</table>

**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**

PictureBox, DropDownListPictureBox, and TreeView controls

**Syntax**

```
controlname.AddPicture ( picturename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the control to which you want to add an icon, cursor, or bitmap to the main image list</td>
</tr>
<tr>
<td>picturename</td>
<td>The name of the icon, cursor, or bitmap you want to add to the main image list</td>
</tr>
</tbody>
</table>

**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 at runtime 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 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:

```power
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)
```

**See also**

DeletePicture
AddSeries

Description
Add 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)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to add a series</td>
</tr>
<tr>
<td>seriesname</td>
<td>A string whose value is the name of the series you want to add to controlname</td>
</tr>
</tbody>
</table>

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`:

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

These statements add an unnamed series to the graph `gr_product_data`:

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

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

Applies to
ListView controls

Syntax
`listviewname.AddSmallPicture ( picturename )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listviewname</code></td>
<td>The name of the ListView control to which you are adding a small image</td>
</tr>
<tr>
<td><code>picturename</code></td>
<td>The name of the bitmap, icon, or cursor you are adding to the ListView control small image list</td>
</tr>
</tbody>
</table>

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.
AddStatePicture

Examples

This example adds the file "shortcut.ico" to the small picture index of the ListView \texttt{lv\_files}:

\begin{verbatim}
//Add small picture
integer index
index = lv_files.AddSmallPicture("shortcut.ico")
\end{verbatim}

See also

DeleteSmallPicture

AddStatePicture

Description

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

Applies to

ListView and TreeView controls

Syntax

\texttt{controlname.AddStatePicture ( picturename )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{controlname}</td>
<td>The name of the ListView or TreeView control to which you are adding a bitmap, cursor, or icon</td>
</tr>
<tr>
<td>\texttt{picturename}</td>
<td>The name of the bitmap, icon, or cursor you are adding</td>
</tr>
</tbody>
</table>

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

Examples

This example adds the file \texttt{star.ico} to the state picture index of the ListView \texttt{lv\_files}:

\begin{verbatim}
//Add state picture
integer index
index = lv_files.AddStatePicture("star.ico")
\end{verbatim}

See also

DeleteStatePicture
AddToLibraryList

Description

Adds new files to the library search path of an application or component at runtime.

Syntax

`AddToLibraryList ( filelist )`

Argument | Description
---|---
`filelist` | A comma-separated list of file names. Specify the full file name 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 or component is being run in the PowerBuilder development environment, rather than from a standalone executable or server.

-2 The new library list or existing library list is empty, or another internal error has occurred.

Usage

When an 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. For a deployed component, PowerBuilder searches the PBD files in the component’s library list. You can specify additional library files with AddToLibraryList.

Calling `AddToLibraryList` appends a new list of files, in the order in which they are specified in `filelist`, to the list of library files specified in the target. If `filelist` contains a file name that is already in the library list, that file name is not added to the library list. If `filelist` contains more than one occurrence of a given file name, the first occurrence is added to the library list.

To avoid problems that can occur when components share resources, you should use `AddToLibraryList` instead of `SetLibraryList` to add additional PBD files to the search list of a component deployed to EAServer.

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 or component needs.

This function has no effect in the PowerBuilder development environment.
Arrange

Examples

This example adds different PBDs to the library search path depending on whether product or customer processing is to be performed:

```powerbuilder
CHOOSE CASE processkind
CASE "product"
    AddToLibraryList(prod.pbd)
CASE "customer"
    AddToLibraryList(cust.pbd)
END CHOOSE
```

See also

GetLibraryList
SetLibraryList

### 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:

```powerbuilder
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)
```

**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:

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

**See also**
GetActiveSheet
OpenSheet
**Asc**

**Description**
Converts the first character of a string to its Unicode code point. A code point is the numerical integer value given to a Unicode character.

**Syntax**

\[
\text{Asc} \left( \text{string} \right)
\]

**Argument** | **Description**
--- | ---
\text{string} | The string for which you want the code point value of the first character

**Return value**
Unsigned Integer. Returns the code point value of the first character in \text{string}. If \text{string} is null, \text{Asc} returns null.

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

**Examples**
This statement returns 65, the code point value for uppercase A:

\[
\text{Asc} \left( \text{"A"} \right)
\]

This example checks if the first character of string \text{ls_name} is uppercase:

\[
\text{String \text{ls_name}}
\]

\[
\text{IF Asc(\text{ls_name}) > 64 and Asc(\text{ls_name}) < 91 \text{ THEN ...}}
\]

See also
AscA
Char
Mid
Asc method for DataWindows in the *DataWindow Reference* or online Help

---

**AscA**

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

**Syntax**

\[
\text{AscA} \left( \text{string} \right)
\]

**Argument** | **Description**
--- | ---
\text{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 \text{string}. If \text{string} is null, AscA returns null.

**Usage**
You can use AscA to find out the case of a character by testing whether its ASCII value is within the appropriate range. A separate function, Asc, is provided to return the Unicode code point of a character.
Examples

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

\[
\text{AscA}(\text{"A"})
\]

This example checks if the first character of string \text{ls\_name} is uppercase:

\[
\text{String \text{ls\_name}} \quad \text{IF AscA(\text{ls\_name}) > 64 \text{ and AscA(\text{ls\_name}) < 91 THEN ...}
\]

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 (\text{iarr}) which is the integer array:

\[
\begin{align*}
\text{string str\_from\_int, hold\_str} \\
\text{integer arraylen} \\
\text{arraylen} = \text{UpperBound(iarr)} \\
\text{FOR i = 1 to arraylen} \\
\quad \text{// Convert first character of pair to a char} \\
\quad \text{hold\_str} = \text{CharA(iarr[i])} \\
\quad \text{// Add characters to string after converting} \\
\quad \text{// the integer's high byte to char} \\
\quad \text{str\_from\_int} += \text{hold\_str} + \& \\
\quad \text{CharA((iarr[i] - \text{AscA(hold\_str)}) / 256)} \\
\text{NEXT}
\end{align*}
\]

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

See also

Asc
CharA
Mid
AscA method for DataWindows in the DataWindow Reference or online Help
**ASin**

**Description**
Calculates the arcsine of an angle.

**Syntax**

```
ASin(n)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The ratio of the lengths of two sides of a triangle for which you want a corresponding angle (in radians). The ratio must be a value between -1 and 1.</td>
</tr>
</tbody>
</table>

**Return value**
Double. Returns the arcsine of n.

**Examples**
This statement returns .999998 (rounded to six places):

```
ASin(.84147)
```

This statement returns .520311 (rounded to six places):

```
ASin(LogTen(Pi(1)))
```

This statement returns 0:

```
ASin(0)
```

This code in the Clicked event of a button catches a runtime error that occurs when an arcsine is taken for a user-entered value—passed in a variable—that is outside of the permitted range:

```powershell
double ld_num
ld_num = double(sle_1.text)
try
    sle_2.text = string(asin(ld_num))
catch (runtimeerror er)
    messagebox("Runtime Error", er.getmessage())
end try
```

**See also**
Sin
ACos
ATan
Pi
ASin method for DataWindows in the *DataWindow Reference* or online Help
**ATan**

**Description**
Calculates the arctangent of an angle.

**Syntax**
```
ATan(n)
```

**Argument** | **Description**
--- | ---

`n` | The ratio of the lengths of two sides of a triangle for which you want a corresponding angle (in radians)

**Return value**
Double. Returns the arctangent of `n`.

**Examples**
This statement returns 0:
```
ATan(0)
```

This statement returns 1.000 (rounded to three places):
```
ATan(1.55741)
```

This statement returns 1.267267 (rounded to six places):
```
ATan(Pi(1))
```

**See also**
Tan
ASin
ACos
ATan method for DataWindows in the *DataWindow Reference* or online Help

---

**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)
```
BeginTransaction

Description
Creates an EAServer transaction and associates it with the calling thread.

Applies to
CORBACurrent objects

Syntax
CORBACurrent.BeginTransaction()

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBACurrent</td>
<td>Reference to the CORBACurrent service instance</td>
</tr>
</tbody>
</table>

Return value
Boolean. Returns true if it succeeds and false if the transaction could not be created.

Usage
The BeginTransaction function creates a transaction and modifies the transaction context of the calling thread so that it is associated with the transaction. This enables the calling thread to obtain information about the transaction and control commits and rollbacks. BeginTransaction can be called by a client or a component that is marked as OTS style. EAServer must be using the two-phase commit transaction coordinator (OTS/XA).

If the calling thread is already associated with a transaction, BeginTransaction returns false. Nested transactions are not supported.

Examples
This example shows the use of BeginTransaction to create a transaction from a client:

```powerbuilder
// Instance variables:
// CORBACurrent corbcurr
// Connection myconnect
long ll_rc
integer li_rcl, li_rc2
boolean lb_success
ll_rc = myconnect.ConnectToServer()
// insert error handling ...
li_rcl = this.GetContextService("CORBACurrent", &
corbcurr)
// insert error handling ...
li_rc2 = corbcurr.Init( myconnect )
// insert error handling ...
lb_success = corbcurr.BeginTransaction()
IF NOT lb_success THEN
    MessageBox ("Create Transaction Failed", &
        "The client may already be in a transaction")
    RETURN
```
ELSE
    ll_rc = myconnect.CreateInstance(lcst_mybookstore)
    // begin processing

See also
CommitTransaction
GetContextService
GetStatus
GetTransactionName
Init
ResumeTransaction
RollbackOnly
RollbackTransaction
SetTimeout
SuspendTransaction

Blob

Converts a string or byte array to a blob.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert a string to a blob</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Convert a string or byte array to a blob</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

**Convert a string to a blob**

**Description**
Converts a string to a blob datatype.

**Syntax**

```
Blob( text [, encoding] )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The string you want to convert to a blob datatype</td>
</tr>
<tr>
<td>encoding</td>
<td>Character encoding of the resulting blob. Values are:</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• EncodingANSI!</td>
</tr>
<tr>
<td>• EncodingUTF8!</td>
</tr>
<tr>
<td>• EncodingUTF16LE!</td>
</tr>
<tr>
<td>• EncodingUTF16BE!</td>
</tr>
</tbody>
</table>

**Return value**
Blob. Returns the converted string in a blob with the requested encoding, if specified. If `text` is null, Blob returns null.
**Blob**

**Usage**
If the `encoding` argument is not provided, Blob converts a Unicode string to a Unicode blob. You must provide the `encoding` argument if the blob has a different encoding.

**Examples**
This example saves a text string as a Unicode blob:

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

This example saves a text string as a blob with UTF-8 encoding:

```powershell
Blob Blob
Blob = Blob("Any Text", EncodingUTF8!)
```

**See also**
BlobEdit
BlobMid
String

**Syntax 2**
Convert a string or a byte array to a blob

**Description**
Converts a string or an array of bytes to a blob datatype.

**Syntax**

```powershell
Blob (array[])
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringorbytearray</td>
<td>An Any variable that holds a string or an array of bytes you want to convert to a blob datatype</td>
</tr>
</tbody>
</table>

**Return value**
Blob. Returns the converted string or byte array in a blob.

**Examples**
This example saves an array of bytes as a blob, then copies the contents of the blob to another byte array:

```powershell
Blob lblb_1
Any a
byte byte_array[], byte_array2[]

// initialize array
byte_array[] = {1,10,100,200,255}

a = byte_array
lblb_1 = Blob(a)

byte_array2[] = GetByteArray(lblb_1)
```

**See also**
GetByteArray
BlobEdit

**Description**
Inserts data of any PowerBuilder datatype into a blob variable.

**Syntax**
```
BlobEdit( blobvariable, n, data {, encoding} )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blobvariable</td>
<td>An initialized variable of the blob datatype into which you want to copy a standard PowerBuilder datatype</td>
</tr>
<tr>
<td>n</td>
<td>The number (1 to 4,294,967,295) of the position in blobvariable at which you want to begin copying the data</td>
</tr>
<tr>
<td>data</td>
<td>Data of a valid PowerBuilder datatype that you want to copy into blobvariable</td>
</tr>
<tr>
<td>encoding</td>
<td>Character encoding of the blob variable in which you want to insert data of datatype string. Values are:</td>
</tr>
<tr>
<td></td>
<td>- EncodingANSI!</td>
</tr>
<tr>
<td></td>
<td>- EncodingUTF8!</td>
</tr>
<tr>
<td></td>
<td>- EncodingUTF16LE! (default)</td>
</tr>
<tr>
<td></td>
<td>- EncodingUTF16BE!</td>
</tr>
</tbody>
</table>

**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.

If the data argument is a string, the position in the blobvariable in which you want to copy data will be the length of the string + 2. If the data argument is a string converted to a blob, the position will be the length of the string + 1. This is because a string contains a null terminating character that it loses when it is converted to a blob. Thus, BlobEdit (blob_var, 1, "ZZZ") returns 5, while BlobEdit (blob_var, 1, blob ("ZZZ")) returns 4.

Use the encoding parameter if the data argument is a string and you want to generate a blob with a specific encoding.

**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()
```
This example copies a string into the blob `blb_data` starting at position 1 and specifies that the blob should use ANSI encoding:

```powershell
blob{100} blb_data
string str1 = "This is a string"
ulong ul_pos
ul_pos = BlobEdit (blb_data, 1, str1, EncodingANSI)
```

See also

Blob
BlobMid

---

### BlobMid

**Description**

Extracts data from a blob variable.

**Syntax**

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data</code></td>
<td>Data of the blob datatype</td>
</tr>
<tr>
<td><code>n</code></td>
<td>The number (1 to 4,294,967,295) of the first byte you want returned</td>
</tr>
<tr>
<td><code>length</code> (optional)</td>
<td>The number of bytes (1 to 4,294,967,295) you want returned</td>
</tr>
</tbody>
</table>

**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 \textit{datablob} starting at position 5 in the blob \textit{data\_1}; the second call stores the bytes of datablob from position 5 to the end in \textit{data\_2}:

\begin{verbatim}
blob data\_1, data\_2, datablob

... // Read a blob datatype into datablob.

data\_1 = BlobMid(datablob, 5, 10)
data\_2 = BlobMid(datablob, 5)
\end{verbatim}

This code copies a bitmap in the blob \textit{emp\_photo} starting at position 1, stores the position at which the next copy can begin in \textit{nbr}, and then copies a date into the blob \textit{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 \textit{st\_1}:

\begin{verbatim}
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)))
\end{verbatim}

See also

Blob
BlobEdit

\section*{BuildModel}

\textbf{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.

\textbf{Applies to}

Profiling and TraceTree objects
### BuildModel

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instancename</code></td>
<td>Instance name of the Profiling or TraceTree object</td>
</tr>
<tr>
<td><code>progressobject</code> (optional)</td>
<td>A PowerObject that represents the number of activities that have been processed</td>
</tr>
<tr>
<td><code>eventname</code> (optional)</td>
<td>A string specifying the name of an event you define</td>
</tr>
<tr>
<td><code>triggerpercent</code> (optional)</td>
<td>A long identifying the number of activities the BuildModel function should process before triggering the <code>eventname</code> event</td>
</tr>
</tbody>
</table>

#### Syntax

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

#### 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 )
```
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:

```power_script
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:

```power_script
TraceTree ltct_model
String ls_filename

ltct_model = CREATE TraceTree
ltct_model.SetTraceFileName(ls_filename)
lct_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:

```power_script
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

**Byte**

**Description**
Converts a number into a Byte datatype or obtains a Byte value stored in a blob.

**Syntax**
```powerbuilder
Byte ( stringorblob )
```

**Argument**
| stringorblob |
| Description |
| A String or any numeric datatype that you want to return as a Byte, or a Blob datatype in which the initial value is the Byte value that you want to return. The stringorblob variable can also have an Any datatype as long as it references a string, integer, uint, long, longlong, or blob. |

**Return value**
Byte. Returns the value of the stringorblob variable as a Byte datatype if it succeeds; it returns 0 if the stringorblob variable is not a valid PowerScript number or if it has an incompatible datatype. If stringorblob is null, Byte returns null.

**Usage**
If the number you convert exceeds the upper range of the Byte datatype (>255), the Byte method returns the difference between the number you pass in the stringorblob argument and the nearest multiple of 256 below that number.

If you pass a blob in the stringorblob argument, only the value of the initial character is converted to a byte value. (There is no “overflow” when you use a blob argument.) To get the byte value for a character at a different position in the blob, you can use the GetByte method.
Examples

This example converts a string entered in a SingleLineEdit control to a byte value:

```powerscript
Byte ly_byte
ly_byte = Byte(sle_1.text)
```

If the text entered in the SingleLineEdit is 4, the byte value of ly_byte is 4. If the text entered is 257, the value of ly_byte is 1. For 256 or text such as “ABC12”, the value of ly_byte is 0.

This example returns the ASCII value of the initial character that you enter in a SingleLineEdit control:

```powerscript
Byte lb_byte
Blob myBlob
myBlob = Blob(sle_1.text)
lb_byte = Byte(myBlob)
```

See also
GetByte
SetByte

---

### Cancel

**Description**
Stops the execution of a pipeline object.

**Applies to**
Pipeline objects

**Syntax**
```
pipelineobject.Cancel()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipelineobject</td>
<td>The name of a pipeline user object that contains the pipeline object to be executed</td>
</tr>
</tbody>
</table>

**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 Users 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
**CancelSync**

**Description**
Cancels the synchronization process and rolls back any changes accumulated during the processing.

**Applies to**
MLSynchronization, MLSync controls

**Syntax**
```
SyncObject.CancelSync()
```

**Argument** | **Description**
--- | ---
`syncObject` | The name of the synchronization object that started a synchronization process that you want to stop.

**Return value**
Integer. Returns 1 for success and -1 for failure.

**Examples**
The following code in the Clicked event of the Cancel button on a wizard-generated synchronization progress window cancels the synchronization:
```
long rc
rc = i_uosync.cancelsync()
IF rc = 1 THEN
    mle_status.text += 'Synchronization Cancelled~r~n'
ELSE
    mle_status.text += 'Cancel request failed.~r~n'
END IF
```

**See also**
Synchronize

---

**CanUndo**

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

**Applies to**
Any editable control (DataWindow, MultiLineEdit, SingleLineEdit, RichTextEdit)

**Syntax**
```
editname.CanUndo()
```

**Argument** | **Description**
--- | ---
`editname` | The name of the DataWindow control, 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:

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

See also  `Undo`
CategoryName

Description
Obtains the category name associated with the specified category number.

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

Syntax
controlname.CategoryName( { graphcontrol, } categorynumber )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to find the name of a specific category, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>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.</td>
</tr>
<tr>
<td>categorynumber</td>
<td>The number of the category for which you want the name.</td>
</tr>
</tbody>
</table>

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:

```powerbuilder
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:

```powerbuilder
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)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The number for which you want the smallest whole number that is greater than or equal to it</td>
</tr>
</tbody>
</table>

Return value
The datatype 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 method for DataWindows in the DataWindow Reference or online Help

ChangeDirectory

Description
Changes the current directory.

Syntax
```
ChangeDirectory (directoryname)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>directoryname</td>
<td>String for the name of the directory you want to set as the current directory</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if the function succeeds and -1 if an error occurs.
ChangeMenu

Examples

This example changes the current directory to the parent directory of the current directory and displays the new current directory in a SingleLineEdit control:

```powershell
ChangeDirectory( ".." )
```

`sle_1.text= GetCurrentDirectory( )`

See also

CreateDirectory
GetCurrentDirectory

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 } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windowname</code></td>
<td>The name of the window for which you want to change the menu.</td>
</tr>
<tr>
<td><code>menuname</code></td>
<td>The name of the menu you want to make the current menu.</td>
</tr>
<tr>
<td><code>position</code></td>
<td>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).</td>
</tr>
</tbody>
</table>

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.

In an MDI frame window, if you change to a menu with a different menu and toolbar style (traditional or contemporary), the style of the menu changes but the style of the toolbar does not.
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 Unicode character of a string or converts an integer to a char.

Syntax

```
Char(n)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>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.</td>
</tr>
</tbody>
</table>

Return value

Char. Returns the first Unicode 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
CharA
**CharA**

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

**Syntax**

`CharA(n)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>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. <code>N</code> can also be an Any variable containing a string, integer, or blob.</td>
</tr>
</tbody>
</table>

**Return value**
`Char`. Returns the first character of `n`. If `n` is null, `CharA` returns null.

**Examples**

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

```
string ls_S
ls_S = CharA(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 = CharA(64 + li_DeliveryNbr)
    ... // Additional processing of ls_Delivery
NEXT
```

**See also**
AscA
Char
Char method for DataWindows in the *DataWindow Reference* or online Help
Check

**Description**
Displays a checkmark next to a menu item in a drop-down or cascading menu and sets the menu item’s Checked property to true.

**Applies to**
Menu objects

**Syntax**
```
mename.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 drop-down 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` drop-down menu on the menu bar `m_Appl`:
```
m_Appl.m_View.m_Grid.Check ()
```

**See also**
`Uncheck`
### ChooseColor

**Description**
Displays the standard color selection dialog box.

**Syntax**
```
ChooseColor ( color {, customcolors []} )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>A long passed by reference that represents the color selected in the dialog box</td>
</tr>
<tr>
<td>customcolors</td>
<td>A long array of custom colors passed by reference to the color selection dialog box</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 1 if the function succeeds, 0 if the user selects cancel (or the dialog box is closed), -1 if an error occurs.

**Examples**
This example displays the color selection dialog box with a base color of red and with two different custom colors defined:

```pascal
long red, green, blue
long custom[ ]
integer li_color
red = 255
custom[1]=rgb(red, 200, blue)
custom[2]=8344736
li_color = ChooseColor( red, custom [ ] )
```

**See also**
RGB

---

### ClassList

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

**Applies to**
Profiling object

**Syntax**
```
instancename.ClassList ( list )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instancename</td>
<td>Instance name of the Profiling object.</td>
</tr>
<tr>
<td>list</td>
<td>An unbounded array variable of datatype ProfileClass in which ClassList stores a ProfileClass object for each class included in the model. This argument is passed by reference.</td>
</tr>
</tbody>
</table>
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:

```power
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]
    ...
NEXT
```

See also

BuildModel

ClassName

Determines the class of an object or the datatype of a variable.

<table>
<thead>
<tr>
<th>To determine</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>The class of an object</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>The class (or datatype) of a variable</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>
**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 datatype, such as Window! or CheckBox!. `ClassName` reports the class of the object in the ancestor-descendant hierarchy.

**Examples**
These statements return the class of the dragged control `Source`:

```
DragObject Source
string which_class

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

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

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

FOR i = 1 TO UpperBound(control[])
    the_object[i] = control[i]
    the_class[i] = the_object[i].ClassName()
NEXT
```
Suppose your object hierarchy has a window named ancestor_window and it has descendants called win1 and win2, and the user can choose which descendant to open as a sheet. The following code tests which descendant window class is currently active (the MDI frame is w_frame):

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

See also
DraggedObject
TypeOf

## Syntax 2

### For variables

Provides the datatype of a variable.

#### Syntax

```
ClassName( variable )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>The name of the variable for which you want to know its name (that is, its datatype)</td>
</tr>
</tbody>
</table>

#### Return value

String. Returns the name of variable. Returns the empty string ("") if variable is an enumerated datatype or if an error occurs. If variable is null, ClassName returns null.

#### Usage

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

#### Examples

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

```
string varname
varname = ClassName(gd_double)
```
Clear

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

Applies to
DataWindow, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, DropDownPictureListbox, OLE controls, and OLEStorage objects

Syntax
```plaintext
objectname.Clear()
```

Return value
Integer for DataWindow, InkEdit, and list boxes, Long for other controls.

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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| `objectname` | One of the following:  
- The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox or DropDownPictureListbox from which you want to delete (clear) selected text.  
- 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. |
Examples

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

```powerscript
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:

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

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

```powerscript
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:

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

See also

Close
Cut
Paste
ReplaceText
SelectText

---

**ClearBoldDates**

**Description**
Clears all bold date settings that had been set with SetBoldDate.

**Applies to**
MonthCalendar control

**Syntax**

```powerscript
controlname.ClearBoldDates()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the MonthCalendar control from which you want to clear the bold dates</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 0 for success and -1 for failure.

**Usage**
You can use the SetBoldDate function to specify that selected dates, such as holidays, display in bold. ClearBoldDates clears all such settings. To clear individual bold dates, use the SetBoldDate function with the `onoff` parameter set to false.
**Examples**

This example clears all bold settings in the control `monthCalVacations`:

```powerbuilder
integer li_return
li_return = monthCalVacation.ClearBoldDates()
```

See also

`SetBoldDate`

---

**Clipboard**

Retrieves or replaces the contents of the system clipboard.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve or replace the contents of the system clipboard with text</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Replace the contents of the system clipboard with a bitmap image of a graph</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For text**

**Description**

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

**Syntax**

`Clipboard ( { string } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string (optional)</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.
**Calling Clipboard in a DataWindow control or DataStore object**  
To retrieve or replace the contents of the system clipboard with text from a DataWindow item (cell value), you must first assign the value to a string and then call the system Clipboard function as follows:

```
string ls_data = dw_1.object.column_name[row_number] ::Clipboard(ls_data)
```

The DataWindow version of Clipboard, documented in Syntax 2 (and in the DataWindow Reference), is only applicable to graphs.

**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 objects in windows and user objects, and graphs in DataWindow controls and DataStore objects.
Close

Syntax

```
name.Clipboard({graphobject})
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the graph or the DataWindow control or DataStore containing the graph you want to copy to the clipboard</td>
</tr>
<tr>
<td>graphobject</td>
<td>A string whose value is the name of the graph in the DataWindow object that you want to copy to the clipboard</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>To close</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A window</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>An OLEStorage object variable, saving the object and clearing connections between it and a storage file or object</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>A stream associated with the specified OLEStream object variable</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>A trace file</td>
<td>Syntax 4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowname</td>
<td>The name of the window you want to close</td>
</tr>
</tbody>
</table>

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. Do not call Close from the CloseQuery or Close events, since this produces an endless loop.

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

Closing a window by calling the Close function in any of the window’s events or in an event of any control on the window can cause PowerBuilder to crash if the Close function is not the last statement in the event script. You can avoid this issue by calling the Close function in the last statement of the event script, or in a user-defined event that is posted from the event script. For example, the following code in the Open event script for a window called w_1 can cause a crash:

```// w_1 Open event script
close(this)
open(w_2) // causes crash```

This code does not cause a crash:

```// w_1 ue_postopen event script
close(this)```

## 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.
Close

Examples

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

```powerbuilder
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:

```powerbuilder
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.

Applies to

OLEStorage objects

Syntax

```powerbuilder
olestorage.Close()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olestorage</td>
<td>The OLEStorage object variable that you want to save and close</td>
</tr>
</tbody>
</table>

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:

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

See also

Open
Save
SaveAs
**Syntax 3**  
For OLEStream objects  
**Description**  
Closes an OLEStream object.  
**Applies to**  
OLEStream objects  
**Syntax**  
`olestream.Close()`  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olestream</code></td>
<td>The OLEStream object variable that you want to close</td>
</tr>
</tbody>
</table>

**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:

```powerscript
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()`  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instancename</code></td>
<td>Instance name of the TraceFile object</td>
</tr>
</tbody>
</table>

**Return value**  
ErrorReturn. Returns one of the following values:

- **Success!** – The function succeeded
- **FileNotFoundException** – A trace file has not been opened
CloseChannel

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:

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

See also
Reset
Open
NextActivity

---

CloseChannel

Description
Closes a DDE channel.

Syntax
`CloseChannel ( handle [, windowhandle ] )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>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.</td>
</tr>
<tr>
<td>windowhandle</td>
<td>(optional) The handle to the PowerBuilder window that is acting as the DDE client.</td>
</tr>
</tbody>
</table>

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`:

```powerscript
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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tabcontrolname</code></td>
<td>The name of the Tab control containing the tab page you want to close</td>
</tr>
<tr>
<td><code>userobjectvar</code></td>
<td>The name of the user object you want to close</td>
</tr>
</tbody>
</table>

**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.
CloseUserObject

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

```
    winodwname.CloseUserObject ( userobjectname )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>winodwname</td>
<td>The name of the window that contains the user object</td>
</tr>
<tr>
<td>userobjectname</td>
<td>The name of the user object you want to close</td>
</tr>
</tbody>
</table>

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:

```powerscript
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:

```powerscript
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 use CloseWithReturn only for response windows.

Applies to

Window objects

Syntax

```powerscript
CloseWithReturn ( windowname, returnvalue )
```

---

383
CloseWithReturn

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowname</td>
<td>The name of the window you want to close.</td>
</tr>
<tr>
<td>returnValue</td>
<td>The value you want to store in the Message object when the window is closed. Returnvalue must be one of these datatypes:</td>
</tr>
<tr>
<td></td>
<td>• String</td>
</tr>
<tr>
<td></td>
<td>• Numeric</td>
</tr>
<tr>
<td></td>
<td>• PowerObject</td>
</tr>
</tbody>
</table>

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. Do not call Close or CloseWithReturn from these events, since this produces an endless loop.

Before executing the event scripts, CloseWithReturn stores returnvalue in the Message object, and 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 results.

PowerBuilder stores returnvalue in the Message object properties according to its datatype. 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 datatype.

Table 10-1: Message object properties where return values are stored

<table>
<thead>
<tr>
<th>Return value datatype</th>
<th>Message object property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>Message.DoubleParm</td>
</tr>
<tr>
<td>PowerObject (such as a structure)</td>
<td>Message.PowerObjectParm</td>
</tr>
<tr>
<td>String</td>
<td>Message.StringParm</td>
</tr>
</tbody>
</table>
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)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The TreeView control in which you want to collapse an item</td>
</tr>
<tr>
<td>itemhandle</td>
<td>The handle of the item you want to collapse</td>
</tr>
</tbody>
</table>

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

**Usage**
If there is only one level-one entry, you can use the RootTreeItem handle as the argument to collapse the tree so that only the top-level node is displayed. However, CollapseItem collapses only the current item, so that if the children of the top-level item were expanded when the tree was collapsed, they will still be expanded when the top-level item is expanded.

If there is more than one level-one entry, using the RootTreeItem handle as the argument collapses only the first level-one entry.

**Examples**
This example collapses an item in a TreeView control:
```
long ll_tvi
ll_tvi = tv_list.FindItem(currenttreetem!, 0)
tv_list.CollapseItem(ll_tvi)
```

This example collapses the top-level item in a TreeView control that has only one level-one entry:
```
long ll_tvi
ll_tvi = tv_list.FindItem(roottreeitem!, 0)
```
CommandParm

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

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 do not need to call CommandParm in the application’s Open event. Use the commandline argument instead.

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

```powerShell
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:

```powerShell
string ls_cmd, ls_arg[]
```
### CommitTransaction

**Description**
Declares that the EAServer transaction associated with the calling thread should be committed.

**Applies to**
CORBACurrent objects

**Syntax**

```
CORBACurrent.CommitTransaction (breportheuristics)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBACurrent</td>
<td>Reference to the CORBACurrent service instance</td>
</tr>
<tr>
<td>breportheuristics</td>
<td>A boolean specifying whether heuristic decisions should be reported for the transaction associated with the calling thread</td>
</tr>
</tbody>
</table>
CHAPTER 10  PowerScript Functions

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

-1  Failed for unknown reason
-2  No transaction is associated with the calling thread
-3  The calling thread does not have permission to commit the transaction
-4  The HeuristicRollback exception was raised
-5  The HeuristicMixed exception was raised
-6  The HeuristicHazard exception was raised

Usage

The CommitTransaction function completes the transaction associated with the calling thread. Use the BeginTransaction function to begin a transaction and associate it with the calling thread. The transaction is not completed if any other participants in the transaction vote to roll back the transaction.

CommitTransaction can be called by a client or a component that is marked as OTS style. EAServer must be using the two-phase commit transaction coordinator (OTS/XA).

Examples

In this example, the client calls the dopayroll method on the CmpnyAcct EAServer component, which processes a company payroll. The method returns 1 if the company has sufficient funds to meet the payroll, and the client then commits the transaction:

```power
define payroll
    // Instance variables:
    // CORBACurrent corbcurr
    integer li_rc
    boolean lb_rv
    long ll_rc

    // Create an instance of the CORBACurrent object
    // and initialize it
    ...
    lb_rv = corbcurr.BeginTransaction()
    IF lb_rv THEN
        ll_rc = myconnect.CreateInstance(CmpnyAcct)
        // handle error
        li_rc = CmpnyAcct.dopayroll()
        IF li_rc = 1 THEN
            corbcurr.CommitTransaction()
        ELSE
            corbcurr.RollbackTransaction()
        END IF
    ELSE
        corbcurr.RollbackTransaction()
    END IF
```


ConnectToNewObject

ELSE
    // handle error
END IF

See also
BeginTransaction
GetContextService
GetStatus
GetTransactionName
Init
ResumeTransaction
RollbackOnly
RollbackTransaction
SetTimeout
SuspendTransaction

ConnectToNewObject

Description
Creates a new 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, OLETxnObject objects

Syntax
oleobject.ConnectToNewObject ( classname )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>The name of an OLEObject variable that you want to connect to an automation server or COM object. You cannot specify an OLEObject that is the Object property of an OLE control.</td>
</tr>
<tr>
<td>classname</td>
<td>A string whose value is a programmatic identifier or class ID that identifies an automation server or COM server.</td>
</tr>
</tbody>
</table>

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
-15 COM+ is not loaded on this computer
-16 Invalid Call: this function not applicable
If any argument’s value is null, `ConnectToNewObject` returns null.

**Usage**

The `OLEObject` variable can be used for automation, in which the PowerBuilder application asks the server application to manipulate the OLE object programmatically. It can also be used to connect to a COM object that is registered on a local or remote computer or that is installed in COM+.

The `OLETxnObject` variable is used to provide COM+ transaction control to PowerBuilder clients. Calling `ConnectToNewObject` with an `OLETxnObject` variable creates a new object instance within the transaction context associated with the variable. If COM+ is not loaded on the client computer, the `ConnectToNewObject` call fails. Use `SetAbort` to abort the transaction or `SetComplete` to complete it if all other participants in the transaction concur.

For more information about automation and connecting to COM objects, see `ConnectToObject`.

**Deprecated support for COM and COM+ components**

Starting with version 11.5, you can no longer deploy COM or COM+ components from the PowerBuilder IDE. However, you can continue to connect to existing PowerBuilder COM and COM+ components from PowerBuilder applications.

**Examples**

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

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

This example creates an `OLETxnObject` variable and calls `ConnectToNewObject` to create and connect to a new instance of a PowerBuilder COM object on a COM+ server:

```powerbuilder
OLETxnObject EmpObj
Integer li_rc
EmpObj = CREATE OLETxnObject
li_rc = EmpObj.ConnectToNewObject("PB110COM.employee")
IF li_rc < 0 THEN
    DESTROY EmpObj
    MessageBox("Connecting to COM Object Failed", &
                "Error: " + String(li_rc))
    Return
END IF
```
// Perform some work with the COM object
...
// If the work completed successfully, commit
// the transaction and disconnect the object
EmpObj.SetComplete();
EmpObj.DisconnectObject();

See also
ConnectToObject
DisconnectObject
SetAbort
SetComplete

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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>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.</td>
</tr>
<tr>
<td>hostname</td>
<td>A string whose value is the name of the remote host where the COM server is located.</td>
</tr>
<tr>
<td>classname</td>
<td>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.</td>
</tr>
</tbody>
</table>

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
-15 COM+ is not loaded on this computer
-16 Invalid Call: this function not applicable to OLETxnObject

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 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:

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>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.</td>
</tr>
<tr>
<td>filename</td>
<td>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.</td>
</tr>
<tr>
<td>classname</td>
<td>(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.</td>
</tr>
</tbody>
</table>

**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 Cannot connect to the currently active object
- -6 Filename is not valid
- -7 File not found or file could not be opened
- -8 Load from file not supported by server
- -9 Other error
- -15 COM+ is not loaded on this computer
- -16 CreateCall: this function not applicable to OLETransactionObject

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 datatype 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 path name):

```
integer result
OLEObject myoleobject

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

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

```powerbuilder
integer result
OLEObject myoleobject

myoleobject = CREATE OLEObject
result = myoleobject.ConnectToRemoteObject("", &"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 })`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>oleobject</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>hostname</code></td>
<td>A string whose value is the name of the remote host where the COM server is located.</td>
</tr>
<tr>
<td><code>filename</code></td>
<td>A string whose value is the name of an OLE storage file. You cannot specify an empty string, COM looks for <code>filename</code> on the local (client) machine. If <code>filename</code> 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, <code>\hostname\shared_directory\test.ext</code>.</td>
</tr>
<tr>
<td><code>classname</code></td>
<td>(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 <code>classname</code> and <code>filename</code>, is an OLE structured storage file, PowerBuilder uses the class ID in <code>filename</code>. Otherwise, PowerBuilder uses the filename extension to determine what server application to start.</td>
</tr>
</tbody>
</table>
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
-15 COM+ is not loaded on this computer
-16 Invalid Call: this function not applicable to OLETxnObject

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 general information about OLE automation, see ConnectToObject.

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:

```plaintext
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:

```plaintext
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 component. The client application must call ConnectToServer before it can use a remote object on the server.

This function applies to distributed applications only.

**Applies to**

Connection objects

**Syntax**

```
connection.ConnectToServer()
```
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 Connection object not connected to server
- 62 Server busy
- 92 Required property is missing or invalid

Usage
Before calling `ConnectToServer`, you assign values to the properties of the Connection object.

Examples
In this example, the client application connects to a server application using the Connection object `myconnect`:

```power_script
// Global variable:
// connection myconnect
long ll_rc
myconnect = create connection
myconnect.driver = "jaguar"
myconnect.location = "Jagserver1:2000"
myconnect.application = "PB_pkg_1"
myconnect.userID = "bjones"
myconnect.password = "mypass"
ll_rc = myconnect.ConnectToServer()
IF ll_rc <> 0 THEN
   MessageBox("Connection failed", ll_rc)
END IF
```

You can enclose the `ConnectToServer` function in a try-catch block to catch exceptions thrown during the attempt to connect. This example uses `SSLServiceProvider` and `SSLCallBack` objects to create a secure connection. An exception or other error in any of the SSLCallback functions raises the CTSSecurity::UserAbortedException. The error-handling code shown in the example displays a message box with the text of the error message, but your code should take additional appropriate action:

```power_script
SSLServiceProvider   sp
```
// set QOP
getcontextservice( "SSLServiceProvider", sp )
sp.setglobalproperty( "QOP", "sybpks_simple" )
// set PB callback handler
sp.setglobalproperty( "CallbackImpl", &
    "uo_sslcallback_handler" )

// connect to the server
connection cxn
 cxn.userid   = "jagadmin"
cxn.password = "sybase"
cxn.driver   = "jaguar"
cxn.application = "dbgpkg"
cxn.options     = "ORBLogFile='d:\PBJagClient.Log'"
cxn.location = "iiops://localhost:9001"

TRY
 l_rc = cxn.ConnectToServer()
CATCH (userabortedexception uae)
    MessageBox("UserAbortedException Caught", &
        "ConnectToServer caught: " + uae.getMessage() )
    l_rc = 999
CATCH ( CORBASystemException cse )
    MessageBox("CORBASystemException Caught", &
        "ConnectToServer caught: " + cse.getMessage() )
    l_rc = 998
CATCH ( RuntimeError re )
    MessageBox("RuntimeError Exception Caught", &
        "ConnectToServer caught: " + re.getMessage() )
    l_rc = 997
CATCH ( Exception ex )
    MessageBox("Exception Caught", &
        "ConnectToServer caught: " + ex.getMessage() )
    l_rc = 996
END TRY

IF l_rc <> 0 THEN
    MessageBox("Error", "Connection Failed - code: " &
        + string(l_rc) )
    MessageBox("Error Info", "ErrorCode= " + &
        string(cxn.ErrCode) + "~nErrText= " + &
        cxn.ErrText)
ELSE
    MessageBox("OK", "Connection Established")
END IF

See also
DisconnectServer

**Copy**

**Description**

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

**Applies to**

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

**Syntax**

```
objectref.Copy()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectref</td>
<td>One of the following:</td>
</tr>
<tr>
<td></td>
<td>- The name of the DataWindow control, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox or DropDownPictureListBox containing the text you want to copy to the clipboard.</td>
</tr>
<tr>
<td></td>
<td>- 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.</td>
</tr>
<tr>
<td></td>
<td>The fully qualified name for a DWObject has this syntax: <code>dwcontrol.Object.dwobjectname</code></td>
</tr>
<tr>
<td></td>
<td>If <code>objectref</code> is a DataWindow, text is copied from the edit control over the current row and column. If <code>objectref</code> is a DropDownListBox or DropDownPictureListBox, its AllowEdit property must be true.</td>
</tr>
</tbody>
</table>

**Return value**

Integer for DataWindow, InkEdit, and list boxes, Long for other controls.

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.
Copy

For OLE controls and OLE DWObejcts, 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:

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

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

```powerbuilder
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

```powerbuilder
rtname.CopyRTF ( { selected {, band} } )
```
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 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:

```powerscript
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:

```powerscript
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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The angle (in radians) for which you want the cosine</td>
</tr>
</tbody>
</table>

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

ACos

Pi

Sin

Tan

Cos method for DataWindows in the DataWindow Reference or online Help
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:

```powerbuilder
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
```

CreateDirectory
Description Creates a directory.
Applies to File system
Syntax CreateDirectory ( directoryname )
Return value Integer. Returns 1 if the function succeeds and -1 if an error occurs.
Examples This example creates a new subdirectory in the current path and then makes the new subdirectory the current directory:

```powerbuilder
string ls_path="my targets"
integer li_filenum
CreateDirectory ( ls_path )
li_filenum = ChangeDirectory( ls_path )
```
See also GetCurrentDirectory
RemoveDirectory
CreateInstance

CreateInstance

Creates an instance of a remote object running on a middle-tier server.

### To create a remote object instance

<table>
<thead>
<tr>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax 1</td>
</tr>
<tr>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1

**For creating an object instance on a remote server**

**Description**

Creates an instance of a component running on EAServer. This function can be used to instantiate a remote object from a PowerBuilder client. In addition, it can be used within a component running on EAServer to instantiate another component running on a different server.

**Applies to**

Connection objects

**Syntax**

`connection.CreateInstance(objectvariable [, classname] )`

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connection</code></td>
<td>The name of the Connection object used to establish the connection.</td>
</tr>
<tr>
<td><code>objectvariable</code></td>
<td>A global, instance, or local variable whose datatype is the same class as the object being created or an ancestor of that class.</td>
</tr>
<tr>
<td><code>classname</code> (optional)</td>
<td>A string whose value is the name of the class datatype to be created. You can optionally prepend a package name followed by a slash to the class name (for example, &quot;mypkg/mycomponent&quot;).</td>
</tr>
</tbody>
</table>

**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**

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.

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:

```powerscript
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
**Syntax 2**

**For creating a component instance on the current server**

**Description**
Creates an instance of a component running on the current EAserver or COM+ server. This function is called from within a component instance running on EAserver or COM+.

**Applies to**
TransactionServer objects

**Syntax**

```
transactionserver.CreateInstance (objectvariable {, classname} )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transactionserver</td>
<td>Reference to the TransactionServer service instance.</td>
</tr>
<tr>
<td>objectvariable</td>
<td>A global, instance, or local variable whose datatype is the same class as the object being created or an ancestor of that class.</td>
</tr>
</tbody>
</table>
| classname (optional) | A string whose value is the name of the class datatype to be created.  
For EAserver components, you can optionally prepend a package name followed by a slash to the class name (for example, "mypackage/mycomponent").  
For COM+ components, you can optionally prepend a ProgID followed by a period to the class name (for example, "PowerBuilder:HTMLDataWindow"). |

**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**
The CreateInstance function on the TransactionServer context object allows you to access other EAserver or COM+ components running on the current server. The created instance inherits all the transaction and security attributes of the current object.
On EAServer, the TransactionServer CreateInstance method invokes the EAServer name service to create proxies. Proxies for remote components might be returned by the name service rather than an instance that is running locally. To guarantee that a locally installed instance is used, specify the component name as "local:package/component", where package is the package name and component is the component name. The call fails if the component is not installed in the same server.

The CreateInstance function on the TransactionServer context object uses the same user and password information that applies to the component from which it is called.

Before you can use the transaction context service, you need to declare a variable of type TransactionServer and call the GetContextService function to create an instance of the service.

Examples

The following statements show how an EAServer component might instantiate another component in the same server and call one of its methods:

```
Integer rc
rc = this.GetContextService("TransactionServer", &
    ts)
IF rc <> 1 THEN
    // handle the error
END IF
rc = this.CreateInstance(mycomp2, &
    "mypackage/nvo_comp2")

IF IsValid(mycomp2) = FALSE THEN
    // handle the error
END IF
mycomp2.method1()
```

This example shows the syntax for creating an instance of a COM component:

```
Integer rc
OleObject lole
TransactionServer lts

lole = create OleObject
rc = this.GetContextService("TransactionServer", lts)
IF rc <> 1 THEN
    return "Error from GetContextService " + String (rc)
END IF

// PBCOM is the ProgID, n_genapp is the class name
rc = lts.CreateInstance(lole, "PBCOM.n_genapp")
```
CreatePage

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

Applies to
User objects used as tab pages

Syntax
userobject.CreatePage()

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 is 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 has not 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:

```powerscript
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`

**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.

**Applies to**
DataWindow, InkEdit, MultiLineEdit, SingleLineEdit, DropDownListBox, DropDownPictureListBox, and OLE controls

**Syntax**

```powerscript
controlname.Cut()
```

**Argument** | **Description**
--- | ---
`controlname` | The name of the DataWindow, InkEdit, 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**
Integer for DataWindow, InkEdit, and list boxes, Long for other controls.

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.
Cut

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()
```
**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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>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.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) The name of the graph in the DataWindow control or DataStore for which you want the data point count for the series. (DataWindow control or DataStore only)</td>
</tr>
<tr>
<td>seriesname</td>
<td>A string whose value is the name of the series for which you want the number of data points.</td>
</tr>
</tbody>
</table>

**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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control for which you want to get data in a DataWindow</td>
</tr>
<tr>
<td>dwsource</td>
<td>The name of the DataWindow control, DataStore, or child DataWindow that contains the data to be connected with input fields in rtename</td>
</tr>
</tbody>
</table>

**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:

```powerShell
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:

```powerShell
DataStore ds_empinfo = CREATE DataStore
ds_empinfo.DataObject = "d_emp"
rs_empinfo.SetTransObject(SQLCA)
rs_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 datatype of the source data.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract the date from DateTime data or extract a date stored in a blob</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Convert a string to a date</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Combine numeric data into a date</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>
**Platform information for Windows**

To make sure you get correct return values for the year, you must verify that yyyy is the Short Date Style for year in the Regional Settings of the user’s Control Panel. Your program can check this with the RegistryGet function.

If the setting is not correct, you can ask the user to change it manually or have the application change it (by calling the RegistrySet function). The user may need to reboot after the setting is changed.

---

**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 )
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datetime</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

**Return value**

Date. Returns the date in `datetime` as a date. If `datetime` contains an invalid date or an incompatible datatype, 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`:

```powershell
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:

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

**See also**

DateTime
Syntax 2

For strings

Description
Converting a string whose value is a valid date to a date value.

Syntax
Date (string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string containing a valid date (such as January 1, 2002, or 12-31-99) that you want returned as a date. Datetime can also be an any variable containing a string.</td>
</tr>
</tbody>
</table>

Return value
Date. Returns the date in string as a date. If string contains an invalid date or an incompatible datatype, 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 datatype date format yyyy-mm-dd.

PowerBuilder attempts to match the input string to a date format in the regional settings on the computer. In PowerBuilder 10 and later, if a complete match is not found, PowerBuilder attempts a partial match. For example, if you use Date('01-JAN-1900') and PowerBuilder finds the partial match (dd-MMM-yy), PowerBuilder parses the first two numbers of the year and gets 19. The 2-digit year is interpreted as a year between 1930 and 2029, and the date returned is 1/1/2019.

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

Examples

Example 1 These statements all return the date datatype for text expressing the date July 4, 2004 (2004-07-04). The system setting for a date’s format is set with the month’s position in the middle:

\[
\begin{align*}
\text{Date} & \quad ("2004/07/04") \\
\text{Date} & \quad ("2004 July 4") \\
\text{Date} & \quad ("04 July 2004")
\end{align*}
\]

Example 2 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 version 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**

```powerbuilder

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 method for DataWindows in the DataWindow Reference or the online Help

### 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 (1 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 datatype. 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 2005-10-15:

```powerbuilder

date ld_my_date
ld_my_date = Date(2005, 10, 15)
```

See also

- DateTime
DaysAfter
RelativeDate
RelativeTime

**DateTime**

Manipulates DateTime values. There are two syntaxes.

<table>
<thead>
<tr>
<th>Syntax 1</th>
<th>Syntax 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>To</td>
<td>Use</td>
</tr>
<tr>
<td>Combine a date and a time value into a DateTime value</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Obtain a DateTime value that is stored in a blob</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For creating DateTime values**

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

**Syntax**

```powerbuild
DateTime (date, time)
```

**Argument**   | **Description**                                                                 |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date</code></td>
<td>A value of type <code>date</code>.</td>
</tr>
<tr>
<td><code>time</code></td>
<td>A value of type <code>time</code>. If you omit <code>time</code>, PowerBuilder sets <code>time</code> to 00:00:00.000000 (midnight). If you specify <code>time</code>, only the hour portion is required. (optional)</td>
</tr>
</tbody>
</table>

**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:

```powerbuild
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 method for DataWindows in the DataWindow Reference or the online Help

### Syntax 2

**For extracting DateTime values from blobs**

**Description**

Extracts a DateTime value from a blob.

**Syntax**

```
DateTime ( blob )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blob</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>A date value from which you want the day</td>
</tr>
</tbody>
</table>
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 2004-01-31 and set li_day_portion to that value:

```powerscript
integer li_day_portion
li_day_portion = Day(2004-01-31)
```

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:

```powerscript
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 method for DataWindows in the DataWindow Reference or the online Help

### DayName

**Description**  Determines the day of the week in a date value and returns the weekday’s name.

**Syntax**  

```powerscript
DayName ( date )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>A date value for which you want the name of the day</td>
</tr>
</tbody>
</table>

**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 runtime files available on the machine where the application is run. If you have installed localized runtime files in the development environment or on a user’s machine, then on that machine the name returned by DayName is in the language of the localized files.

For information about localized runtime files, which are available in French, German, Italian, Spanish, Dutch, Danish, Norwegian, and Swedish, see Application Techniques.

**Examples**

These statements evaluate the date literal 2003-07-04 and set day_name to Sunday:

```powershell
string day_name
day_name = DayName(2003-07-04)
```

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

```powershell
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**

```powershell
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 2000-01-31 and set day_nbr to 4 (January 31, 2000, was a Wednesday):

```
integer day_nbr
day_nbr = DayNumber(2000-01-31)
```

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 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  \textbf{DaysAfter} (date1, date2)

\begin{tabular}{|l|l|}
\hline
\textbf{Argument} & \textbf{Description} \\
\hline
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 \\
\hline
\end{tabular}

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(2002-12-20, 2002-12-24)
```
This statement returns -4:

\texttt{DaysAfter(2002-12-24, 2002-12-20)}

This statement returns 0:

\texttt{DaysAfter(2003-12-24, 2003-12-24)}

This statement returns 5:

\texttt{DaysAfter(2003-12-29, 2004-01-03)}

If you declare \texttt{date1} and \texttt{date2} date variables and assign February 16, 2003, to \texttt{date1} and April 28, 2003, to \texttt{date2} as follows:

\begin{verbatim}
  date date1, date2
\end{verbatim}

\begin{verbatim}
  date1 = 2003-02-16
  date2 = 2003-04-28
\end{verbatim}

then each of the following statements returns 71:

\begin{verbatim}
  DaysAfter(date1, date2)
  DaysAfter(2003-02-16, date2)
  DaysAfter(date1, 2003-04-28)
  DaysAfter(2003-02-16, 2003-04-28)
\end{verbatim}

See also

RelativeDate
RelativeTime
SecondsAfter
DaysAfter in the \textit{DataWindow Reference}
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 does not support, you can use DBHandle to provide the handle as an argument to one of these external functions.

Examples
For examples, search for DBHandle 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()
**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**

```PowerScript
controlname.DeleteCategory (categoryvalue)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The graph in which you want to delete a category.</td>
</tr>
<tr>
<td>categoryvalue</td>
<td>A value that is the category you want to delete from controlname. The value you specify must be the same datatype as the datatype of the category axis.</td>
</tr>
</tbody>
</table>

**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 datatype, `Dec` returns 0. If `stringorblob` is null, `Dec` returns null.

**Examples**

This statement returns 24.3 as a decimal datatype:

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

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

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

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

**See also**

Double

Integer

Long

Real

---

**DeleteCategory**

**Argument**

`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.

**Description**

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 datatype, `Dec` returns 0. If `stringorblob` is null, `Dec` returns null.

**Examples**

This statement returns 24.3 as a decimal datatype:

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

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

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

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

**See also**

Double

Integer

Long

Real
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:

```power
string CategName
CategName = sle_delete.Text
g_r_product_data.DeleteCategory(CategName)
```

See also  DeleteData
          DeleteSeries

---

**DeleteColumn**

Description  Deletes a column.

ListView controls

Syntax  
```
listviewname.DeleteColumn( index )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control from which you want to delete a column</td>
</tr>
<tr>
<td>index</td>
<td>The index number of the column you want to delete</td>
</tr>
</tbody>
</table>

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:

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

See also  DeleteColumns

---

**DeleteColumns**

Description  Deletes all columns.

Applies to  ListView controls

Syntax  
```
listviewname.DeleteColumns()  
```
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to delete a data value</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number of the series containing the data value you want to delete from controlname</td>
</tr>
<tr>
<td>datapointnumber</td>
<td>The number of the data point containing the data you want to delete</td>
</tr>
</tbody>
</table>

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
**DeleteItem**

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

**Syntax 1** For ListBox and DropDownListBox controls

**Description**
Deletes an item from the list of values for a list box control.

**Applies to**
ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

**Syntax**

`listboxname.DeleteItem(index)`

<table>
<thead>
<tr>
<th><strong>Argument</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listboxname</code></td>
<td>The name of the ListBox, DropDownListBox, PictureListBox, or DropDownPictureListBox from which you want to delete an item</td>
</tr>
<tr>
<td><code>index</code></td>
<td>The position number of the item you want to delete</td>
</tr>
</tbody>
</table>

**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.

**Examples**
Assuming `lb_actions` contains 10 items, this statement deletes item 5 from `lb_actions` and returns 9:

`lb_actions.DeleteItem(5)`
DeleteItem

These statements delete the first selected item in lb_actions:

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

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

```powershell
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
```powershell
listviewname.DeleteItem( index )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control from which you want to delete an item</td>
</tr>
<tr>
<td>index</td>
<td>The index number of the item you want to delete</td>
</tr>
</tbody>
</table>

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:

```powershell
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**  
```powerscript
treeviewname.DeleteItem (itemhandle)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The name of the TreeView control from which you want to delete an item</td>
</tr>
<tr>
<td>itemhandle</td>
<td>The handle of the item you want to delete</td>
</tr>
</tbody>
</table>

**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:

```powerscript
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:

```powerscript
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**  
```powerscript
listviewname.DeleteItems()
```
DeleteLargePicture

Description
Deletes a picture from the large image list.

Applies to
ListView controls

Syntax
```
listviewname.DeleteLargePicture(index)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control to which you want to delete a large picture from the image list</td>
</tr>
<tr>
<td>index</td>
<td>The index entry for the large picture you want to delete</td>
</tr>
</tbody>
</table>

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()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control from which you want to delete all pictures from the large picture image list</td>
</tr>
</tbody>
</table>

432
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The control from which you want to delete a picture</td>
</tr>
<tr>
<td>index</td>
<td>The index number of the picture you want to delete from the TreeView control’s image list</td>
</tr>
</tbody>
</table>

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 does not 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: AddPicture, DeletePictures

**DeletePictures**

Description: Deletes all pictures from an image list.

Applies to: PictureListBox, DropDownPictureListBox, and TreeView controls

Syntax: 
```
controlname.DeletePictures ()
```


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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The graph in which you want to delete a series</td>
</tr>
<tr>
<td><code>seriesname</code></td>
<td>A string whose value is the name of the series you want to delete from <code>controlname</code></td>
</tr>
</tbody>
</table>

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 list box 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 list box:
```
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control from which you want to delete a small picture from the image list</td>
</tr>
<tr>
<td>index</td>
<td>The index number of the small picture you want to delete</td>
</tr>
</tbody>
</table>

**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:

```
1v_list.DeleteSmallPicture(1)
```

**See also**
DeleteSmallPictures

---

### DeleteSmallPictures

**Description**
Deletes all small pictures from a ListView control.

**Applies to**
ListView controls

**Syntax**

```
listviewname.DeleteSmallPictures ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control from which you want to delete all small pictures</td>
</tr>
</tbody>
</table>

**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:

```
1v_list.DeleteSmallPictures( )
```

**See also**
DeleteSmallPicture
DeleteStatePicture

Description
Deletes a state picture from a control.

Applies to
ListView and TreeView controls

Syntax
\texttt{controlname.DeleteStatePicture\ (index)}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{controlname}</td>
<td>The name of the ListView or TreeView control from which you want to delete a picture from the state image list</td>
</tr>
<tr>
<td>\texttt{index}</td>
<td>The index number of the state picture you want to delete</td>
</tr>
</tbody>
</table>

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:
\begin{verbatim}
lv_list.DeleteStatePicture(1)
\end{verbatim}

See also
DeleteStatePictures

DeleteStatePictures

Description
Deletes all state pictures from a control.

Applies to
ListView and TreeView controls

Syntax
\texttt{controlname.DeleteStatePictures\ ()}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{controlname}</td>
<td>The name of the ListView or TreeView control from which you want to delete all state pictures</td>
</tr>
</tbody>
</table>

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:
\begin{verbatim}
lv_list.DeleteStatePictures()
\end{verbatim}

See also
DeleteStatePicture
CHAPTER 10  PowerScript Functions

DestroyModel
Description  Destroys the current performance analysis or trace tree model.
Applies to  Profiling and TraceTree objects
Syntax  \[ instancename.DestroyModel() \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ instancename ]</td>
<td>Instance name of the Profiling or TraceTree object</td>
</tr>
</tbody>
</table>

Return value  ErrorReturn. Returns one of the following values:
- Success! – The function succeeded
- ModelNotExistError! – 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:

\[
\text{lpro\_model.}\text{DestroyModel}()
\]

DESTROY lpro\_model

See also  BuildModel

DirectoryExists
Description  Determines if the named directory exists.
Syntax  DirectoryExists(\[ directoryname \])

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ directoryname ]</td>
<td>String for the name of the directory you want to verify as existing</td>
</tr>
</tbody>
</table>

Return value  Returns true if the directory exists. Returns false if the directory does not exist or if you pass a file name in the \[ directoryname \] argument.

Usage  You can use this method before attempting to move a file or delete a directory using other file system methods.
Examples

This example determines if a directory exists before attempting to move a file to it; otherwise it displays a message box indicating that the path does not exist:

```powerbuilder
string ls_path="monthly targets"

If DirectoryExists ( ls_path ) Then
    FileMove ("2000\may.csv", ls_path+"\may.csv"
    MessageBox ("File Mgr", "File moved to "+
        + ls_path + ".")
Else
    MessageBox ("File Mgr", "Directory " + ls_path+'&
        " does not exist"
End If
```

See also

FileMove
GetCurrentDirectory
RemoveDirectory

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, DirList can display the current drive and directory as well.

Applies to

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>The name of the ListBox control you want to populate.</td>
</tr>
<tr>
<td>filespec</td>
<td>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.</td>
</tr>
</tbody>
</table>
CHAPTER 10  PowerScript Functions

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 cannot be populated (for example, filespec is a file, not a directory, or specifies 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 list box 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.
DirSelect

Examples

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

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

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

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

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

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

See also

DirSelect
GetFolder

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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>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.</td>
</tr>
<tr>
<td>selection</td>
<td>A string variable in which the selected path name will be put.</td>
</tr>
</tbody>
</table>

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 file name:

```power_script
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`
`GetFolder`

---

**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 ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuname</td>
<td>The name of the menu selection you want to deactivate (disable)</td>
</tr>
</tbody>
</table>

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:

\[
\text{m\_appl.m\_edit.Enabled} = \text{FALSE}
\]

is equivalent to:

\[
\text{m\_appl.m\_edit.Disable()}
\]

**Examples**

This statement disables the m\_edit menu item on the menu m\_appl:

\[
\text{m\_appl.m\_edit.Disable()}
\]

**See also**

Enable

---

**DisableCommit**

**Description**

Declares that a component’s transaction updates are inconsistent and cannot be committed in their present state.

** Applies to**

TransactionServer objects

**Syntax**

\[
\text{transactionserver.DisableCommit()}
\]

**Argument** | **Description**
---|---
\text{transactionserver} | Reference to the TransactionServer service instance

**Return value**

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

**Usage**

The DisableCommit function indicates that the current transaction cannot be committed because the component’s work has not been completed; the instance remains active after the current method returns. The DisableCommit function corresponds to the disallowCommit transaction primitive in EAServer.

**Examples**

The following example shows the use of the DisableCommit in a component method that performs database updates:

```powerbuilder
// Instance variables:
// DataStore ids_datastore
// TransactionServer ts
Integer li_rc
long ll_rv

li_rc = this.GetContextService("TransactionServer", & ts)
IF li_rc <> 1 THEN
    // handle the error
END IF
```
ll_rv = ids_datastore.Update()

IF ll_rv = 1 THEN
    ts.EnableCommit()
ELSE
    ts.DisableCommit()
END IF

See also
EnableCommit
IsInTransaction
IsTransactionAborted
SetAbort
SetComplete
Which

DisconnectObject

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

Applies to
OLEObject objects

Syntax
oleobject.DisconnectObject()

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 more information about OLE automation, see ConnectToObject.
### DisconnectServer

#### Examples

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

```powerbuilder
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.

#### Applies to

Connection objects

#### Syntax

```
connection.DisconnectServer ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>The name of the Connection object used to establish the connection you want to delete</td>
</tr>
</tbody>
</table>

#### 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`:

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

See also

ConnectToServer

---

**Double**

**Description**

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

**Syntax**

```powerscript
Double ( stringorblob )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stringorblob</code></td>
<td>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. <code>stringorblob</code> can also be an Any variable containing a double or blob.</td>
</tr>
</tbody>
</table>

**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 datatype, `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:

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

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

```powerscript
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):

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

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

Description
Requests the OLE server application to execute the specified verb for the OLE object in an OLE control or OLE DWObject.

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 cannot 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:

```powerbuilder
integer result
result = ole_1.DoVerb(7)
```
This example executes verb 7 for the object in the OLE DObject ole_graph:

```powerscript
integer result
result = dw_1.Object.ole_graph.DoVerb(7)
```

See also
- Activate
- OLEActivate in the *DataWindow Reference*
- 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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>control</code></td>
<td>The name of the control you want to drag or stop dragging</td>
</tr>
<tr>
<td><code>dragmode</code></td>
<td>A value of the DragMode datatype indicating the action you want to take on control:</td>
</tr>
<tr>
<td></td>
<td>- Begin! – Put <code>control</code> in drag mode</td>
</tr>
<tr>
<td></td>
<td>- Cancel! – Stop dragging <code>control</code> but do not cause a DragDrop event</td>
</tr>
<tr>
<td></td>
<td>- End! – Stop dragging <code>control</code> and if <code>control</code> is over a target object, cause a DragDrop event</td>
</tr>
</tbody>
</table>

**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(Begin!) in a control’s Clicked event to manually put the 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.

**Dragging DataWindow controls**
The Clicked event of a DataWindow control occurs when the user presses the mouse button, not when the mouse button is released. If you place Drag(Begin!) in a DataWindow control’s Clicked event, releasing the mouse button ends the drag. To achieve the same behavior as with other controls, define a user-defined event for the DataWindow control called lbuttonup and map it to the pbm_lbuttonup event ID. Then place the following code in the lbuttonup event script (ib_dragflag is a boolean instance variable):

```plaintext
IF NOT ib_dragflag THEN
    this.Drag(Begin!)
    ib_dragflag = TRUE
ELSE
    ib_dragflag = FALSE
END IF
```

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:

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

**See also**
DraggedObject

---

**DraggedObject**

**Description**
Returns a reference to the control that triggered a drag event.
Obsolete function
You no longer need to call the DraggedObject function in a drag event. Use the event’s source argument instead.

Syntax
DraggedObject ()

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

No control
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 datatype of the control. To access the properties of the control, you can assign the DragObject reference to a variable of that control’s datatype (see the example).

Examples
These statements set which_control equal to the datatype 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.

<table>
<thead>
<tr>
<th>To enable editing of a label in a</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The ListView control in which you want to enable label editing</td>
</tr>
<tr>
<td>index</td>
<td>The index of the ListView item to be edited</td>
</tr>
</tbody>
</table>

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

Usage  
The EditLabels property for the ListView must be set to true to enable editing of labels. When this property is true, calling the EditLabel function sets focus on the item and enables editing. To disable editing when the user has finished editing the label, set the EditLabels property to false in the EndLabelEdit event.

If the EditLabels property is set to false, the EditLabel function does not enable editing.

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

```powerscript
integer li_selected
li_selected = lv_1.SelectedIndex()
lv_1.EditLabels = TRUE
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
Enable

Syntax

treeviewname. EditLabel ( itemhandle )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The TreeView control in which you want to enable label editing</td>
</tr>
<tr>
<td>itemhandle</td>
<td>The handle of the item to be edited</td>
</tr>
</tbody>
</table>

Return value

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

Usage

The EditLabels property for the TreeView must be set to true to enable editing of labels. When this property is true, calling the EditLabel function sets focus on the item and enables editing. To disable editing when the user has finished editing the label, set the EditLabels property to false in the EndLabelEdit event.

If the EditLabels property is set to false, the EditLabel function does not enable editing.

Examples

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

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuname</td>
<td>The name of the menu selection you want to enable</td>
</tr>
</tbody>
</table>

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

---

**EnableCommit**

**Description**

Declares that a component's work may be incomplete but its transaction updates are consistent and can be committed.

**Applies to**

TransactionServer objects

**Syntax**

```
transactionserver.EnableCommit()
```

**Argument** | **Description**
--- | ---
`transactionserver` | Reference to the TransactionServer service instance

**Return value**

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

**Usage**

The EnableCommit function indicates that the component should not be deactivated after the current method invocation. However, if the component instance is deactivated, the current transaction can be committed. The EnableCommit function corresponds to the continueWork transaction primitive in EAServer.

**Examples**

The following example shows the use of EnableCommit in a component method that performs database updates:

```powerscript
// Instance variables:
// DataStore ids_datastore
// TransactionServer ts
Integer li_rc
long ll_rv

li_rc = this.GetContextService("TransactionServer",ts)
IF li_rc <> 1 THEN
    // handle the error
END IF
```
EntryList

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

Applies to
TraceTree objects

Syntax
instancename.EntryList( list )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instancename</td>
<td>Instance name of the TraceTree object.</td>
</tr>
<tr>
<td>list</td>
<td>An unbounded array variable of datatype TraceTreeNode in which EntryList stores a TraceTreeNode object for each top-level entry in the trace tree model. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

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:

```powerscript
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

---

**ExecRemote**

Asks a DDE server application to execute the specified command.

<table>
<thead>
<tr>
<th>To send</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single command to a DDE server application (a cold link)</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>A command to a DDE server application after you have opened a channel (a warm link)</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**  
For sending single commands

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

Syntax:  
```
ExecRemote (command, applname, topicname)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>command</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
ExecRemote

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applname</td>
<td>A string whose value is the DDE name of the server application.</td>
</tr>
<tr>
<td>topicname</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 to 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 acknowledges 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 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):

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

### See also

CloseChannel
GetRemote
OpenChannel
SetRemote
StartHotLink
CHAPTER 10  PowerScript Functions

**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.

Syntax

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>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.</td>
</tr>
<tr>
<td>handle</td>
<td>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.</td>
</tr>
<tr>
<td>windowhandle (optional)</td>
<td>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 do not specify windowhandle, the active window acts as the DDE client.</td>
</tr>
</tbody>
</table>

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 **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 more information, see the StartHotLink function.
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:

```powerbuilder
long handle

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

... // Some processing
ExecRemote("[Save]", handle)
CloseChannel(handle, Handle(Parent))
```

**Exp**

**Description**

 Raises $e$ to the specified power.

**Syntax**

`Exp (n)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>The power to which you want to raise $e$ (2.71828)</td>
</tr>
</tbody>
</table>

**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:

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

**See also**

Log
LogTen

Exp method for DataWindows in the *DataWindow Reference* or online Help.
### ExpandAll

**Description**
Recursively expands a specified item.

**Applies to**
TreeView controls

**Syntax**
`treeviewname.ExpandAll (itemhandle)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>treeviewname</code></td>
<td>The TreeView control in which you want to expand an item and all the subordinate items in its hierarchy</td>
</tr>
<tr>
<td><code>itemhandle</code></td>
<td>The handle of the item you want to expand</td>
</tr>
</tbody>
</table>

**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:

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

**See also**
CollapseItem
ExpandItem
FindItem

### ExpandItem

**Description**
Expands a specified item.

**Applies to**
TreeView controls

**Syntax**
`treeviewname.ExpandItem (itemhandle)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>treeviewname</code></td>
<td>The TreeView control in which you want to expand an item</td>
</tr>
<tr>
<td><code>itemhandle</code></td>
<td>The handle of the item you want to expand</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 1 if it succeeds and -1 if an error occurs.
ExpandItem expands only a single item. To expand a specified item including its children, use ExpandAll.

Examples

This example expands the current level of a TreeView:

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

See also

CollapseItem

ExpandAll

FindItem

---

**Fact**

Description

Determines the factorial of a number.

Syntax

```powerbuilder
Fact( n )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The number for which you want the factorial</td>
</tr>
</tbody>
</table>

Return value

Double. Returns the factorial of \( n \). If \( n \) is null, Fact returns null.

Examples

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

```powerbuilder
Fact(4)
```

Both these statements return 1:

```powerbuilder
Fact(1)
```

```powerbuilder
Fact(0)
```

See also

Fact method for DataWindows in the *DataWindow Reference* or online Help

---

**FileClose**

Description

Closes the file associated with the specified file number. The file number was assigned to the file with the FileOpen function.
CHAPTER 10  PowerScript Functions

**FileClose**

**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.

**Usage**

The file is saved in the encoding format in which it was opened.

**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:

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

**See also**

FileLength
FileOpen
FileReadEx
FileWriteEx

---

**FileCopy**

**Description**

Copies one file to another, optionally overwriting the target file.

**Syntax**

```
FileCopy ( sourcefile, targetfile {, replace} )
```

**Argument** | **Description**
--- | ---
sourcefile | String for the name of the file you want to copy
targetfile | String for the name of the file you are copying to
replace (optional) | Boolean specifying whether to replace the target file (true) or not (false)

**Return value**

Integer. Returns values as follows:

- 1 – Success
- -1 – Error opening sourcefile
- -2 – Error writing targetfile

---

PowerScript Reference  461
**FileDelete**

If you do not specify a fully qualified path for *sourcefile* or for *targetfile*, the function works relative to the current directory. If you do not specify the *replace* argument, the FileCopy function does not replace a file in the target directory that has the same name as the name you specify in the *targetfile* argument (This is equivalent to setting the *replace* value to false).

**Examples**

The following example copies a file from the current directory to a different directory and saves the return value in a variable. It does not replace a file of the same name if one already exists in the target directory:

```powershell
integer li_FileNum
li_FileNum = FileCopy (*jazz.gif*, & "C:\emusic\jazz.gif", FALSE)
```

**See also**

FileMove
GetCurrentDirectory

---

### Description

Deletes the named file.

### Syntax

FileDelete ( filename )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>A string whose value is the name of the file you want to delete</td>
</tr>
</tbody>
</table>

### 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:

```powershell
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
FileEncoding

Description
Checks the encoding of the specified file.

Syntax
FileEncoding ( filename )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>The name of the file you want to test for encoding type</td>
</tr>
</tbody>
</table>

Return value
A value of the enumerated datatype Encoding. Values are:
- EncodingANSI!
- EncodingUTF8!
- EncodingUTF16LE!
- EncodingUTF16BE!

If filename does not exist, returns null.

Usage
Use this function to determine the encoding used in an external file before attempting to use it in a PowerBuilder application.

Examples
The following example opens a file in stream mode and tests to determine whether it uses ANSI encoding. If it does, it reads data from the file into a blob and uses the String function to convert the blob to a Unicode string:

```powerbuilder
long ll_filenum
integer li_bytes
string ls_unicode
blob lb_ansi
encoding eRet

ll_filenum = FileOpen("employee.dat",StreamMode!,
                    Read!, LockWrite!, Replace!)

// test the file’s encoding
eRet = FileEncoding("employee.dat")

if eRet = EncodingANSI! then
    li_ bytes = FileReadEx(ll_filenum, lb_ansi)
    ls_unicode = string(lb_ansi, EncodingANSI!)
else
    li_ bytes = FileReadEx(ll_filenum, ls_unicode)
end if
FileClose(ll_filenum)
```
See also Blob
FileClose
FileOpen
FileReadEx
FileWriteEx
String

FileExists

Description
Reports whether the specified file exists.

Syntax
```plaintext
FileExists ( filename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>A string whose value is the name of a file</td>
</tr>
</tbody>
</table>

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 the file can be overwritten:

```plaintext
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 whose size does not exceed 2GB in bytes.

**Syntax**

```
FileLength ( filename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>A string whose value is the name of the file for which you want to know the length. If <code>filename</code> is not on the current application library search path, you must specify the fully qualified name.</td>
</tr>
</tbody>
</table>

**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 bytes at a time.

The length returned by `FileLength` always includes the byte-order mark (BOM). For example, suppose the hexadecimal display of the file `SomeFile.txt` is `FF FE 54 00 68 00 69 00 73 00`, then the following statement returns 10, which includes the BOM:

```
ll_length = FileLength("SomeFile.txt")
```

**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 get a sharing violation.

The `FileLength` function cannot return the length of files whose size exceeds 2GB. Use `FileLength64` to find the length of larger files.

**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", & TextMode!, Read!, LockReadWrite!)
```

The examples for `FileRead` illustrate reading files of different lengths.
**FileLength64**

**Description**
Reports the length of a file of any size in bytes.

**Syntax**

```powerbuilder
FileLength64 ( filename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>A string whose value is the name of the file for which you want to know the length. If <code>filename</code> is not on the current application library search path, you must specify the fully qualified name.</td>
</tr>
</tbody>
</table>

**Return value**
Longlong. Returns the length in bytes of the file identified by `filename`. If the file does not exist, `FileLength64` returns -1. If `filename` is null, `FileLength64` returns `null`.

**Usage**
Call `FileLength64` 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 bytes at a time. Use the `FileReadEx` function to read longer files.

The length returned by `FileLength64` always includes the byte-order mark (BOM). For example, suppose the hexadecimal display of the file `SomeFile.txt` is `FF FE 54 00 68 00 69 00 73 00`, then the following statement returns `10`, which includes the BOM:

```powerbuilder
ll_length = FileLength64("SomeFile.txt")
```

**File security**
If any security is set for the file (for example, if you are sharing the file on a network), you must call `FileLength64` before `FileOpen` or after `FileClose`. Otherwise, you get a sharing violation.

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

```powerbuilder
FileLength64("EMPLOYEE.DAT")
```
These statements determine the length of the EMP.TXT file in the EAST directory and open the file:

```powerscript
long LengthA
integer li_FileNum
LengthA = FileLength64("C:\EAST\EMP.TXT")
li_FileNum = FileOpen("C:\EAST\EMP.TXT", &
    LineMode!, Read!, LockReadWrite!)
```

The examples for FileRead illustrate reading files of different lengths.

See also
FileClose
FileLength
FileOpen
FileReadEx
FileWriteEx

---

**FileMove**

**Description**
Moves a file.

**Syntax**
```
FileMove ( sourcefile, targetfile )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourcefile</td>
<td>String for the name of the file you want to move</td>
</tr>
<tr>
<td>targetfile</td>
<td>String for the name of the location you are moving the file</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns values as follows:

- 1 – Success
- -1 – Error opening sourcefile
- -2 – Error writing targetfile

**Usage**
You cannot write to a target file if a file with the same name already exists in the target directory. If you want to copy over a target file, you can use FileCopy and set the replace argument to true.

**Examples**
This example moves a file from the current directory to a different directory and saves the return value in the li_FileNum variable:

```powerscript
integer li_FileNum
li_FileNum = FileMove ("june.csv", &
    "H:/project/june2000.csv")
```

See also
FileCopy
GetCurrentDirectory
FileOpen

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

Syntax
FileOpen ( filename {, filemode {, fileaccess {, filelock {, writemode { encoding }}}}})

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>A string whose value is the name of the file you want to open. If filename is not on the current directory’s relative search path, you must enter the fully qualified name.</td>
</tr>
<tr>
<td>filemode</td>
<td>(optional) A value of the FileMode enumerated type that specifies how the end of a file read or file write is determined. Values are:</td>
</tr>
<tr>
<td></td>
<td>• LineMode! – (Default) Read or write the file a line at a time</td>
</tr>
<tr>
<td></td>
<td>• StreamMode! – Read blocks of binary data</td>
</tr>
<tr>
<td></td>
<td>• TextMode! – Read text blocks</td>
</tr>
<tr>
<td></td>
<td>For more information, see Usage below.</td>
</tr>
<tr>
<td>fileaccess</td>
<td>(optional) A value of the FileAccess enumerated type that specifies whether the file is opened for reading or writing. Values are:</td>
</tr>
<tr>
<td></td>
<td>• Read! – (Default) Read-only access</td>
</tr>
<tr>
<td></td>
<td>• Write! – Write-only access</td>
</tr>
<tr>
<td></td>
<td>If PowerBuilder does not find the file, a new file is created if the fileaccess argument is set to Write!</td>
</tr>
<tr>
<td>filelock</td>
<td>(optional) A value of the FileLock enumerated type specifying whether others have access to the opened file. Values are:</td>
</tr>
<tr>
<td></td>
<td>• LockReadWrite! – (Default) Only the user who opened the file has access</td>
</tr>
<tr>
<td></td>
<td>• LockRead! – Only the user who opened the file can read it, but everyone has write access</td>
</tr>
<tr>
<td></td>
<td>• LockWrite! – Only the user who opened the file can write to it, but everyone has read access</td>
</tr>
<tr>
<td></td>
<td>• Shared! – All users have read and write access.</td>
</tr>
<tr>
<td>writemode</td>
<td>(optional) A value of the WriteMode enumerated datatype. When fileaccess is Write!, specifies whether existing data in the file is overwritten. Values are:</td>
</tr>
<tr>
<td></td>
<td>• Append! – (Default) Write data to the end of the file</td>
</tr>
<tr>
<td></td>
<td>• Replace! – Replace all existing data in the file</td>
</tr>
<tr>
<td></td>
<td>Writemode is ignored if the fileaccess argument is Read!</td>
</tr>
</tbody>
</table>
CHAPTER 10  PowerScript Functions

PowerScript Reference 469

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  The mode in which you open a file determines the behavior of the functions used to read and write to a file. There are two functions that read data from a file: FileRead and FileReadEx, and two functions that write data to a file: FileWrite and FileWriteEx. FileRead and FileWrite have limitations on the amount of data that can be read or written and are maintained for backward compatibility. They do not support text mode. For more information, see FileRead and FileWrite.

The support for reading from and writing to blobs and strings for the FileReadEx and FileWriteEx functions depends on the mode. The following table shows which datatypes are supported in each mode.

Table 10-2: FileReadEx and FileWriteEx datatype support by mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Blob</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>Not supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Stream</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>Text</td>
<td>Supported</td>
<td>Supported</td>
</tr>
</tbody>
</table>

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

When a file has been opened in stream mode or text mode, FileReadEx reads the whole file until it encounters an EOF or until it reaches a length specified in an optional parameter. FileWriteEx writes the full contents of the string or blob or until it reaches a length specified in an optional parameter.

The optional length parameter applies only to blob data. If the length parameter is provided when the datatype of the second parameter is string, the code will not compile.
In all modes, PowerBuilder can read ANSI, UTF-16, and UTF-8 files. The behavior in stream and text modes is very similar. However, stream mode is intended for use with binary files, and text mode is intended for use with text files. When you open an existing file in stream mode, the file’s internal pointer, which indicates the next position from which data will be read, is set to the first byte in the file.

A byte-order mark (BOM) is a character code at the beginning of a data stream that indicates the encoding used in a Unicode file. For UTF-8, the BOM uses three bytes and is EF BB BF. For UTF-16, the BOM uses two bytes and is FF FE for little endian and FE FF for big endian.

When you open an existing file in text mode, the file’s internal pointer is set based on the encoding of the file:

- If the encoding is ANSI, the pointer is set to the first byte
- If the encoding is UTF-16LE or UTF-16BE, the pointer is set to the third byte, immediately after the BOM
- If the encoding is UTF-8, the pointer is set to the fourth byte, immediately after the BOM

If you specify the optional encoding argument and the existing file does not have the same encoding, FileOpen returns -1.

File not found
If PowerBuilder does not find the file, it creates a new file, giving it the specified name, if the fileaccess argument is set to Write!. If the argument is not set to Write!, FileOpen returns -1.

If the optional encoding argument is not specified and the file does not exist, the file is created with ANSI encoding.

When you create a new text file using FileOpen, use line mode or text mode. If you specify the encoding parameter, the BOM is written to the file based on the specified encoding.

When you create a new binary file using stream mode, the encoding parameter, if provided, is ignored.
Examples

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

```power
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!):

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

This example creates a new file that uses UTF8 encoding. The file is called new.txt and is in the D:\temp directory. It is opened in text mode with write-only access, and no other user can read or write to the file:

```power
integer li_ret
string ls_file
ls_file = "D:\temp\new.txt"
li_ret = FileOpen(ls_file, TextMode!, Write!, &
    LockReadWrite!, Replace!, EncodingUTF8!)
```

See also

FileClose
FileVersion64
FileRead
FileReadEx
FileWrite
FileWriteEx

**FileRead**

**Description**

Reads data from the file associated with the specified file number, which was assigned to the file with the FileOpen function. FileRead is maintained for backward compatibility. Use the FileReadEx function for new development.
**FileRead**

Syntax

```
FileRead ( file#, variable )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file#</td>
<td>The integer assigned to the file when it was opened</td>
</tr>
<tr>
<td>variable</td>
<td>The name of the string or blob variable into which you want to read the data</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns the number of 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. If the file length is greater than 32,765 bytes, FileRead returns 32,765.

Usage

FileRead can read files with ANSI, UTF-8, UTF-16LE, and UTF-16BE encoding.

If the file is an ANSI or UTF-8 file and is read into a string, FileRead converts the text to Unicode before saving it in the string variable. No conversion is needed for UTF-16 files. For Unicode files, the BOM is not written to the string.

If the file is read into a blob, FileRead saves the contents of the file with no conversion. For Unicode files, the BOM is not written to the blob in text mode, but it is written to the blob in stream mode.

If the file was 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 second argument is a blob, FileRead returns -1.

If the file was opened in text mode, FileRead returns -1. Use FileReadEx to read a file in text mode.

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 bytes at a time. Therefore, before calling the FileRead function, call the FileLength64 function to check the file length. If your system has file sharing or security restrictions, you might need to call FileLength64 before you call FileOpen. Use FileReadEx to read longer files.
An end-of-file mark is a null character (ASCII value 0). Therefore, if the file being read contains null characters, FileRead stops reading at the first null character, interpreting it as the end of the 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 = FileLength64("C:\HR\EMP_DATA.TXT")
li_FileNum = FileOpen("C:\HR\EMP_DATA.TXT", &LineMode!)
IF ll_FLength < 32767 THEN
  FileRead(li_FileNum, ls_Emp_Input)
END IF
```

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)
```

See also

FileClose
FileLength64
FileOpen
FileReadEx
FileSeek64
FileWriteEx

**FileReadEx**

**Description**

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

**Syntax**

```
FileReadEx ( file#, blob, { length } )
FileReadEx ( file#, string )
```
**FileReadEx**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file#</td>
<td>The integer assigned to the file when it was opened.</td>
</tr>
<tr>
<td>blob or string</td>
<td>The name of the string or blob variable into which you want to read the data.</td>
</tr>
<tr>
<td>length</td>
<td>In text or stream mode, the number of bytes a retrieve requires. The default value is the length of the file.</td>
</tr>
</tbody>
</table>

**Return value**

Long. Returns the number of bytes read. If an end-of-file mark (EOF) is encountered before any characters are read, FileReadEx returns -100. If the file is opened in LineMode and a CR or LF is encountered before any characters are read, FileReadEx returns 0. If an error occurs, FileReadEx returns -1. FileReadEx returns -1 if you attempt to read from a string in stream mode or read from a blob in line mode. If any argument's value is null, FileReadEx returns null.

**FileReadEx returns long**

Unlike the FileRead function that it replaces, the FileReadEx function returns a long value.

**Usage**

FileReadEx can read files with ANSI, UTF-8, UTF-16LE, and UTF-16BE encoding.

If the file is opened in line mode, FileReadEx 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.

The optional length parameter applies only to blob data. If the length parameter is provided when the datatype of the second parameter is string, the code will not compile.

If the file was opened in stream or text mode, FileReadEx reads to the end of the file or the next length bytes, whichever is shorter. FileReadEx 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 length bytes, FileReadEx automatically positions the pointer after each read operation so that it is ready to read the next chunk of data.

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

If the file is an ANSI or UTF-8 file and is read into a string, FileReadEx converts the text to Unicode before saving it in the string variable. The BOM is not written to the string.
If the file is an ANSI or UTF-8 file and is read into a blob, FileReadEx saves the contents of the file with no conversion. The BOM is not written to the blob in text mode, but it is written to the blob in stream mode.

If the file is in Unicode, no conversion is required.

Examples

This example reads the file EMP_DATA.TXT into a string in text mode. If the file is not in Unicode format, its contents, apart from the BOM, are converted to Unicode and written to the string:

```powerscript
integer li_FileNum
string ls_Emp_Input
li_FileNum = FileOpen("C:\HR\EMP_DATA.TXT", &
TextMode!)
    FileReadEx(li_FileNum, ls_Emp_Input)
END IF
```

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 ll_bytes:

```powerscript
integer li_fnum
long ll_bytes
blob Emp_Id_Pic
li_fnum = FileOpen("C:\HR\EMP_PIC1.BMP", &
StreamMode!)
ll_bytes = FileReadEx(li_fnum, Emp_Id_Pic)
```

See also

FileClose
FileLength64
FileOpen
FileRead
FileSeek64
FileWriteEx

FileSeek

Description

Moves the file pointer to the specified position in a file whose size does not exceed 2GB. The file pointer is the position in the file at which the next read or write begins.
**FileSeek**

**Syntax**

```
FileSeek ( file#, position, origin )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file#</td>
<td>The integer assigned to the file when it was opened.</td>
</tr>
<tr>
<td>position</td>
<td>A long whose value is the new position of the file pointer relative to the position specified in origin, in bytes.</td>
</tr>
<tr>
<td>origin</td>
<td>The value of the SeekType enumerated datatype specifying where you want to start the seek. Values are:</td>
</tr>
<tr>
<td></td>
<td>• FromBeginning! – (Default) At the beginning of the file</td>
</tr>
<tr>
<td></td>
<td>• FromCurrent! – At the current position</td>
</tr>
<tr>
<td></td>
<td>• FromEnd! – At the end of the file</td>
</tr>
</tbody>
</table>

**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 `FileReadEx` or `FileWriteEx` occurs at that position within the file.

If `origin` is set to `FromBeginning!`, and the file is not opened in stream mode, the byte-order mark is ignored automatically. For example, suppose the file’s hexadecimal display is `FF FE 54 00 68 00 69 00 73 00`, the following example illustrates the behavior:

```powerbuilder
long ll_pos
// after the following statement, the file pointer is
// at 68, not 54, and ll_pos = 2, not 4
ll_pos = FileSeek( filenum, 2, FromBeginning!)

// ll_pos = 2, not 4
ll_pos = FileSeek( filenum, 0, FromCurrent!)

// ll_pos = 2, not 4
ll_pos = FileSeek( filenum, -6, FromEnd!)
```

The `FileSeek` function cannot handle files whose size exceeds 2GB. Use `FileSeek64` to move the file pointer in larger files.

**Examples**

This example positions the file pointer 14 bytes from the end of the file:

```powerbuilder
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!:

```powerScript
integer li_FileNum
li_FileNum = FileOpen("emp_data")
FileSeek(li_FileNum, 14, FromCurrent!)
```

See also
FileReadEx
FileSeek64
FileWriteEx

### FileSeek64

**Description**
Moves the file pointer to the specified position in a file of any size. The file pointer is the position in the file at which the next read or write begins.

**Syntax**

```powerScript
FileSeek64 ( file#, position, origin )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file#</td>
<td>The integer assigned to the file when it was opened.</td>
</tr>
<tr>
<td>position</td>
<td>A long whose value is the new position of the file pointer relative to the position specified in origin, in bytes.</td>
</tr>
<tr>
<td>origin</td>
<td>The value of the SeekType enumerated datatype specifying where you want to start the seek. Values are:</td>
</tr>
<tr>
<td></td>
<td> FromBeginning! – (Default) At the beginning of the file</td>
</tr>
<tr>
<td></td>
<td> FromCurrent! – At the current position</td>
</tr>
<tr>
<td></td>
<td> FromEnd! – At the end of the file</td>
</tr>
</tbody>
</table>

**Return value**
Longlong. Returns the file position after the seek operation has been performed. If any argument’s value is null, FileSeek64 returns null.

**Usage**
Use FileSeek64 to move within a binary file that you have opened in stream mode. FileSeek64 positions the file pointer so that the next FileReadEx or FileWriteEx occurs at that position within the file.

If origin is set to FromBeginning!, and the file is not opened in stream mode, the byte-order mark is ignored automatically. For example, suppose the file’s hexadecimal display is FF FE 54 00 69 00 69 00 73 00, the following example illustrates the behavior:

```powerScript
long ll_pos
```
// after the following statement, the file pointer is
// at 68, not 54, and ll_pos = 2, not 4
ll_pos = FileSeek64( filenum, 2, FromBeginning!)

// ll_pos = 2, not 4
ll_pos = FileSeek64( filenum, 0, FromCurrent!)

// ll_pos = 2, not 4
ll_pos = FileSeek64( filenum, -6, FromEnd!)

Examples

This example positions the file pointer 14 bytes from the end of the file:

```powerbuilder
integer li_FileNum
li_FileNum = FileOpen("emp_data")
FileSeek64(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!:

```powerbuilder
integer li_FileNum
li_FileNum = FileOpen("emp_data")
FileSeek64(li_FileNum, 14, FromCurrent!)
```

See also

FileReadEx
FileSeek
FileWriteEx

**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. FileWrite is
maintained for backward compatibility. Use the FileWriteEx function for new
development.

**Syntax**

```powerbuilder
FileWrite( file#, variable )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file#</td>
<td>The integer assigned to the file when the file was opened</td>
</tr>
<tr>
<td>variable</td>
<td>A string or blob whose value is the data you want to write to the file</td>
</tr>
</tbody>
</table>

**Return value**

Integer. Returns the number of 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 can write to files with ANSI, UTF-8, UTF-16LE, and UTF-16BE encoding. 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.

If the data is in a string and the associated file uses ANSI or UTF-8 encoding, FileWrite converts the string to ANSI or UTF-8 encoding before saving it to the associated file.

The behavior of the FileWrite function when the file is opened with the `EncodingANSI!` parameter or with no encoding parameter is platform dependent. On the Windows and Solaris platforms, FileWrite does not convert multilanguage characters to UTF-8 and saves the file with ANSI encoding. On the Linux platform, if the string contains multilanguage characters, FileWrite converts the multi-language characters to UTF-8 and saves the file with UTF-8 encoding.

If the file is opened in stream mode, no conversion is done.

If the file was opened in text mode, FileWrite returns -1. Use FileWriteEx to write to files in text mode.

**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 bytes, FileWrite writes the first 32,765 bytes and returns 32,765. Use FileWriteEx to handle variables that have more than 32,765 bytes.

**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:

```powerscript
integer li_FileNum
li_FileNum = FileOpen("C:\HR\EMP_DATA.TXT", &
  LineMode!, Write!, LockWrite!, Append!)
FileWrite(li_FileNum, "New Employees")
```
The following example reads a blob from the database and writes it to a file. The SQL SELECT statement assigns the picture data to the blob `Emp_Id_Pic`. Then FileOpen opens a file for writing in stream mode and FileWrite writes the blob to the file. You could use the Len function to test whether the blob was too big for a single FileWrite call:

```powerbuilder
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
FileLength64
FileOpen
FileRead
FileReadEx
FileSeek64
FileWriteEx

---

**FileWriteEx**

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

```
FileWriteEx( file#, blob {, length})
```

```
FileWriteEx ( file#, string )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file#</td>
<td>The integer assigned to the file when the file was opened</td>
</tr>
<tr>
<td>blob or string</td>
<td>A blob or string whose value is the data you want to write to the file.</td>
</tr>
<tr>
<td>length</td>
<td>In text or stream mode, the number of bytes to be written. The default value is the length of the file.</td>
</tr>
</tbody>
</table>

Return value

Long. Returns the number of bytes written if it succeeds and -1 if an error occurs. FileWriteEx returns -1 if you attempt to write to a string in stream mode or to a blob in line mode. If any argument’s value is null, FileWriteEx returns null.
FILEWRITEEX returns long
Unlike the FILEWRITE function that it replaces, the FILEWRITEEX function returns a long value.

Usage
FILEWRITEEX can write to files with ANSI, UTF-8, UTF-16LE, and UTF-16BE encoding.

FILEWRITEEX 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 FILEWRITEEX, 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.

FILEWRITEEX sets the file pointer following the last character written. If the file was opened in line mode, FILEWRITEEX writes a carriage return (CR) and linefeed (LF) after the last character in variable and places the file pointer after the CR and LF.

If the file was opened in stream or text mode, FILEWRITEEX writes the full contents of the string or blob or the next length bytes, whichever is shorter. The optional length parameter applies only to blob data. If the length parameter is provided when the datatype of the second parameter is string, the code will not compile.

If the data is in a string and the associated file uses ANSI or UTF-8 encoding, FILEWRITEEX converts the string to ANSI or UTF-8 encoding before saving it to the associated file. If the file is opened in stream mode, no conversion is done.

If the file does not have a byte-order mark (BOM) it is created automatically.

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:

```powerlanguage
integer li_FileNum
li_FileNum = FileOpen("C:\HR\EMP_DATA.TXT", &
                   TextMode!, Write!, LockWrite!, Append!)
FILEWRITEEX(li_FileNum, "New Employees")
```

The following example reads a blob from the database and writes it to a file. The SQL SELECT statement assigns the picture data to the blob Emp_Id_Pic. Then FileOpen opens a file for writing in stream mode and FILEWRITEEX 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:

```powerlanguage
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!)
    FileWriteEx(li_FileNum, emp_id_pic)

See also
FileClose
FileLength64
FileOpen
FileReadEx
FileSeek64

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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chars</td>
<td>A string whose value will be repeated to fill the return string</td>
</tr>
<tr>
<td>n</td>
<td>A long whose value is the length of the string you want returned</td>
</tr>
</tbody>
</table>

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)
FillA

Description
Builds a string of the specified length in bytes by repeating the specified characters until the result string is long enough.

Syntax
FillA (chars, n)

Return value
String. Returns a string n bytes long filled with the characters in the argument chars. If the argument chars has more than n bytes, the first n bytes of chars are used to fill the return string. If the argument chars has fewer than n bytes, the characters in chars are repeated until the return string has n bytes. If any argument's value is null, FillA returns null.

Usage
FillA replaces the functionality that Fill had in DBCS environments in PowerBuilder 9.

In SBCS environments, Fill, FillW, and FillA return the same results.

FillW

Description
Builds a string of the specified length by repeating the specified characters until the result string is long enough. This function is obsolete. It has the same behavior as Fill in SBCS and DBCS environments.

Syntax
FillW (chars, n)

Find

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

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

For syntax for DataWindows and DataStores, see the Find method for DataWindows in the DataWindow Reference or online Help.

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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the RichTextEdit, DataWindow control, or DataStore whose contents you want to search.</td>
</tr>
<tr>
<td>searchtext</td>
<td>A string whose value is the text you want to find. For the RichTextEdit control, <code>searchtext</code> is limited to 99 characters.</td>
</tr>
<tr>
<td>forward</td>
<td>A boolean value indicating the direction you want to search. Values are:</td>
</tr>
<tr>
<td></td>
<td>• TRUE – The search proceeds forward from the cursor position or, if <code>cursor</code> is false, from the start of the document.</td>
</tr>
<tr>
<td></td>
<td>• FALSE – The search proceeds backward from the cursor position or, if <code>cursor</code> is false, from the end of the document.</td>
</tr>
<tr>
<td>insensitive</td>
<td>A boolean value indicating the search string and the found text must match case. Values are:</td>
</tr>
<tr>
<td></td>
<td>• TRUE – The search is not sensitive to case.</td>
</tr>
<tr>
<td></td>
<td>• FALSE – The search is case-sensitive.</td>
</tr>
<tr>
<td>wholeword</td>
<td>A boolean value indicating that the found text must be a whole word. Values are:</td>
</tr>
<tr>
<td></td>
<td>• TRUE – The found text must be a whole word.</td>
</tr>
<tr>
<td></td>
<td>• FALSE – The found text can be a partial word.</td>
</tr>
<tr>
<td>cursor</td>
<td>A boolean value indicating where the search begins. Values are:</td>
</tr>
<tr>
<td></td>
<td>• TRUE – The search begins at the cursor position.</td>
</tr>
<tr>
<td></td>
<td>• FALSE – The search begins at the start of the document if <code>forward</code> is true or at the end if <code>forward</code> is false.</td>
</tr>
</tbody>
</table>

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:

```powerScript
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

**Syntax**

```
ccontrolname.FindCategory ( { graphcontrol, } categoryvalue )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>A string whose value is the name of the graph in which you want to find a specific category, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control in which you want to find a specific category.</td>
</tr>
<tr>
<td>categoryvalue</td>
<td>A value that is the category for which you want the number. The value you specify must be the same datatype as the datatype of the category axis.</td>
</tr>
</tbody>
</table>

**Return value**

Integer. Returns the number of the category named in categoryvalue in the graph controlname, or if controlname is a DataWindow control, in graphcontrol. If an error occurs, FindCategory returns -1. 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 know only a category’s label or when the numbering may have changed.
**FindClassDefinition**

**Examples**

These statements obtain the number of a category in the graph `gr_prod_data`. The category name is the text in the SingleLineEdit `sle_ctgry`:

```powershell
integer CtgryNbr
    CtgryNbr = gr_prod_data.FindCategory(sle_ctgry.Text)
```

These statements obtain the number of the category named Qty in the graph `gr_computers` in the DataWindow control `dw_equip`:

```powershell
integer CtgryNbr
    CtgryNbr = dw_equip.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**

```powershell
FindClassDefinition ( classname {, librarylist } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>classname</code></td>
<td>The name of an object (also called a class or class definition) for which you want information.</td>
</tr>
<tr>
<td><code>librarylist</code></td>
<td>An array of strings whose values are the fully qualified pathnames of PBLs. If you omit <code>librarylist</code>, <code>FindClassDefinition</code> searches the library list associated with the running application.</td>
</tr>
</tbody>
</table>

**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*:

```powerbuilder
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*:

```powerbuilder
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

```powerbuilder
FindFunctionDefinition ( functionname {, librarylist } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>functionname</td>
<td>The name of a global function for which you want information.</td>
</tr>
<tr>
<td>librarylist</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
FindItem

Examples

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

```powerbuilder
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`:

```powerbuilder
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

FindItem

Finds the next item in a list.

<table>
<thead>
<tr>
<th>To find the next item</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>In a ListBox, DropDownListBox, PictureListBox, or DropDownPictureListBox</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>In a ListView control based upon its label</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>By relative position in a ListView control</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>By relative position in a TreeView control</td>
<td>Syntax 4</td>
</tr>
</tbody>
</table>

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  

\[ \text{listboxname}.\text{FindItem}( \text{text}, \text{index} ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>The name of the ListBox control in which you want to find an item.</td>
</tr>
<tr>
<td>text</td>
<td>A string whose value is the starting text of the item you want to find.</td>
</tr>
<tr>
<td>index</td>
<td>The number of the item just before the first item to be searched. To search the whole list, specify 0.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Index number</th>
<th>Item text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Open files</td>
</tr>
<tr>
<td>2</td>
<td>Close files</td>
</tr>
<tr>
<td>3</td>
<td>Copy files</td>
</tr>
<tr>
<td>4</td>
<td>Delete files</td>
</tr>
</tbody>
</table>

Then these statements start searching for Delete starting with item 2 (Close files). FindItem sets Index to 4:

```powershell
    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  

\[ \text{listviewname}.\text{FindItem}( \text{startIndex}, \text{label}, \text{partial}, \text{wrap} ) \]
FindItem

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:

```powershell
definitions
listviewitem 1_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 )
```

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

**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.
**FindItem**

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:

```pascal
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**

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.
`navigationcode` | A value of the TreeNavigation enumerated datatype specifying the relationship between `itemhandle` and the item you want to find. See the table in Usage note for a list of valid values.
`itemhandle` | A long for the handle of an item related via `navigationcode` to the item for which you are searching.

**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. CurrentItem! is not changed until after the clicked event occurs. To return the correct handle for the current item when the user clicks it, create a custom event to return the handle and post it in the clicked event.

If navigationcode is RootTreeItem!, FirstVisibleTreeItem!, CurrentTreeItem!, or DropHighlightTreeItem!, set itemhandle to 0.

The following table shows valid values for the navigationcode argument.

<table>
<thead>
<tr>
<th>Navigationcode value</th>
<th>What FindItem finds</th>
</tr>
</thead>
<tbody>
<tr>
<td>RootTreeItem!</td>
<td>The first item at level 1. Returns -1 if no items have been inserted into the control.</td>
</tr>
<tr>
<td>NextTreeItem!</td>
<td>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.</td>
</tr>
<tr>
<td>PreviousTreeItem!</td>
<td>The sibling before itemhandle. Returns -1 if there are no more siblings.</td>
</tr>
<tr>
<td>ParentTreeItem!</td>
<td>The parent of itemhandle. Returns -1 if the item is at level 1.</td>
</tr>
<tr>
<td>ChildTreeItem!</td>
<td>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.</td>
</tr>
<tr>
<td>FirstVisibleTreeItem!</td>
<td>The first item visible in the control, regardless of level. The position of the scroll bar determines the first visible item.</td>
</tr>
<tr>
<td>NextVisibleTreeItem!</td>
<td>The next expanded item after itemhandle, regardless of level. The NextVisible and PreviousVisible values allow you to walk through all the visible children and branches of an expanded node. Returns -1 if the item is the last expanded item in the control. To scroll to an item that is beyond the reach of the visible area of the control, use FindItem and then SelectItem.</td>
</tr>
<tr>
<td>PreviousVisibleTreeItem!</td>
<td>The next expanded item before itemhandle, regardless of level. Returns -1 if the item is the first root item.</td>
</tr>
</tbody>
</table>
FindMatchingFunction

**Navigationcode value | What FindItem finds**
---|---
CurrentTreeItem! | The selected item. Returns -1 if the control never had focus and nothing has been selected.
DropHighlightTreeItem! | 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.

**Examples**
To return the correct handle when the current item is clicked, place this code in a custom event that is posted in the item’s clicked event:

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

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

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>classdefobject</td>
<td>The name of the ClassDefinition object describing the class in which you want to find a function.</td>
</tr>
</tbody>
</table>
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:

```powerscript
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)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scriptname</td>
<td>A string whose value is the name of the function.</td>
</tr>
<tr>
<td>argumentlist</td>
<td>An unbounded array of strings whose values are the datatypes of the function arguments. If the variable is passed by reference, the string must include &quot;ref&quot; before the datatype. If the variable is an array, you must include array brackets after the datatype. The format is: <code>{ ref } datatype {}</code> For a bounded array, the argument must include the range, as in: <code>ref integer[1 TO 10]</code></td>
</tr>
</tbody>
</table>
FindNext

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

// Datatype 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

\[
\text{controlname}\.\text{FindNext}() \\
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the RichTextEdit or DataWindow control whose contents you want to search</td>
</tr>
</tbody>
</table>

Return value Integer. Returns the number of characters found. FindNext 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 \(\text{rte}_1\) for text the user specifies in the SingleLineEdit \(\text{sle}_\text{search}\). The search proceeds forward from the cursor position, is case insensitive, and is not limited to whole words:

```powerShell
\begin{verbatim}
integer \text{li_charsfound} \\
\text{li_charsfound} = \text{rte}_1\.\text{Find}(\text{sle}_\text{search}.\text{Text}, \&TRUE, \text{TRUE}, \text{FALSE}, \text{TRUE})
\end{verbatim}
```

A second button labeled FindNext would have a script like this:

```powerShell
\begin{verbatim}
\text{rte}_1\.\text{FindNext}()
\end{verbatim}
```

See also Find

FindSeries

Description Obtains the number of a series in a graph when you know the series’ name.

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

Syntax

\[
\text{controlname}\.\text{FindSeries}(\{ \text{graphcontrol}, \} \text{seriesname}) \\
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph containing the series for which you want the number, or the name of the DataWindow control containing the graph</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control containing the series</td>
</tr>
<tr>
<td>(DataWindow control only)</td>
<td></td>
</tr>
<tr>
<td>seriesname</td>
<td>A string whose value is the name of the series for which you want the number</td>
</tr>
</tbody>
</table>

PowerScript Reference 497
Return value

Integer. Returns the number of the series named in `seriesname` in the graph `controlname`, or if `controlname` is a DataWindow control, in `graphcontrol`. If an error occurs, `FindSeries` returns -1. 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 know only 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`:

```powershell
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`:

```powershell
integer SeriesNbr
SeriesNbr = &
    dw_equipment.FindSeries("gr_computers", "PCs")
```

See also

AddSeries
DeleteSeries
FindCategory
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 is 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
datatype. It checks the category of the type definition and, since it is 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:

```
TypeDefnition 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:

TypeDefination td_windef
string ls_libraries[ ]

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

    td_windef = FindTypeDefinition("w_genapp_frame", ls_libraries)

See also
FindClassDefinition
FindFunctionDefinition
FindMatchingFunction

FromAnsi

Description
Converts a blob containing an ANSI character string to a Unicode string.

Syntax
FromAnsi ( blob )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blob</td>
<td>A blob containing an ANSI character string you want to convert to a Unicode string</td>
</tr>
</tbody>
</table>

Return value
String. Returns a character string if it succeeds and an empty string if it fails.

Usage
The FromAnsi function converts an ANSI character string contained in a blob to a Unicode character string.

FromAnsi has the same result as String(blob, EncodingANSI!) and will be obsolete in a future release of PowerBuilder.
**/fromUnicode/**

Unicode files sometimes have two extra bytes at the start of the file to indicate that they are Unicode files.

See also  
FromUnicode  
String  
ToAnsi  
ToUnicode

**FromUnicode**

Description: Converts a blob containing a Unicode character string to a string in the file format of the current version of PowerBuilder.

Syntax:  

```
FromUnicode ( blob )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blob</td>
<td>A blob containing a Unicode character string you want to convert to a string in the file format of the current version of PowerBuilder</td>
</tr>
</tbody>
</table>

Return value: String. Returns a character string if it succeeds and an empty string if it fails.

Usage: The `FromUnicode` function converts a Unicode blob to a Unicode character string and has the same result as `String(blob)`. This function will be obsolete in a future release of PowerBuilder.

Unicode file format

Unicode files sometimes have two extra bytes at the start of the file to indicate that they are Unicode files. If you are opening a Unicode file in stream mode, skip the first two bytes if they are present.

See also  
FromAnsi  
ToAnsi  
ToUnicode
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 to 1 second and sets the variable OldTime to the length of the previous interval:

```powerscript
long OldTime, NewTime
NewTime = 1000 /* 1 second */

OldTime = GarbageCollectSetTimeLimit(NewTime)
```

See also
GarbageCollect
GarbageCollectGetTimeLimit
GetActiveSheet

Description
Returns the currently active sheet in an MDI frame window.

Applies to
MDI frame windows

Syntax
`mdiframewindow.GetActiveSheet()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mdiframewindow</code></td>
<td>The MDI frame window for which you want the active sheet</td>
</tr>
</tbody>
</table>

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:

```powerbuilder
// Declare variable for active sheet
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 \texttt{rtename.GetAlignment()} 

Return value Alignment. A value of the Alignment enumerated datatype 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:

\begin{verbatim}
alignment l_align
l_align = rte_1.GetAlignment()
\end{verbatim}

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 \texttt{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 do not have to code the actual name of the application when you want to set application properties.
GetArgElement

To change whether Toolbar Tips are displayed, you can get the handle of the application object and set the ToolbarTips property:

```pascal
application app
app = GetApplication()
app.ToolbarTips = FALSE
```

The previous example could be coded more simply as follows:

```pascal
GetApplication().ToolbarTips = FALSE
```

---

**GetArgElement**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>activexcontrol</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>Integer specify the argument to return.</td>
</tr>
</tbody>
</table>

**Examples**

This JavaScript example calls the GetArgElement function:

```javascript
... 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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>The name of an OLEObject variable containing the object for which you want the native pointer.</td>
</tr>
<tr>
<td>pointer</td>
<td>An UnsignedLong variable in which you want to store the pointer. If GetAutomationNativePointer cannot get a valid pointer, pointer is set to 0.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage
Pointer is a pointer to OLE’s IUnknown interface. You can use it with the OLE QueryInterface function 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.

This function is useful only for external DLL calls. It is not related to the SetAutomationPointer 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:

```powerbuilder
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

GetByte

Description
Extracts data of type Byte from a blob variable.

Syntax
GetByte ( blobvariable, n, b )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blobvariable</td>
<td>A variable of the Blob datatype from which you want to extract a value of the Byte datatype</td>
</tr>
<tr>
<td>n</td>
<td>The number of the position in blobvariable at which you want to retrieve a value of the Byte datatype</td>
</tr>
<tr>
<td>b</td>
<td>Variable of the Byte datatype in which you want to store the returned data of type Byte</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if it succeeds or -1 if n exceeds the scope of blobvariable; it returns null if the value of any of its arguments is null.

Usage
If you want to get the value of the initial character in a blob, you can use the Byte function without using an argument defining the position of the character.

Examples
This example converts the text in a SingleLineEdit to a blob before obtaining the byte value of the character at the third position:

```powerbuilder
Int li_rtn
Byte lb_byte
Blob myBlob
myBlob = Blob (sle_1.text, EncodingUTF8!)
li_rtn = GetByte(myBlob, 3, lb_byte)
messagebox("getbyte", string(lb_byte))
```
GetByteArray

Description
Obtains an array of Byte values stored in a blob.

Syntax
\texttt{GetByteArray(\hspace{1em}input)}

Return value
Any. Returns the value of the \textit{input} variable as an array of Byte datatypes if it succeeds; it returns 0 if the \textit{input} variable is not a valid blob.

Usage
The returned value can be assigned directly to a byte array.

Examples
This example converts a blob passed in an argument to an array of bytes:

\begin{verbatim}
Byte ly_byte[
ly_byte[] = GetByteArray(blobarg)
\end{verbatim}

See also
Blob
GetByte

GetCertificateLabel

Description
Called by EAServer to allow the user to select one of the available SSL certificate labels for authentication. This function is used by PowerBuilder clients connecting to EAServer.

Applies to
SSLCallBack objects

Syntax
\texttt{sslcallback.GetCertificateLabel(\hspace{1em}thesessioninfo, labels)}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{sslcallback}</td>
<td>An instance of a customized SSLCallBack object.</td>
</tr>
<tr>
<td>\textit{thesessioninfo}</td>
<td>A CORBAObject that contains information about the SSL session. This information can optionally be displayed to the user to provide details about the session.</td>
</tr>
<tr>
<td>\textit{labels}</td>
<td>An array of string values that contains the available certificate labels. The user must select one of these labels.</td>
</tr>
</tbody>
</table>

See also
Byte
SetByte
GetCertificateLabel

Return value
String. Returns one of the labels passed to the function.

Usage
A PowerBuilder application does not usually call the GetCertificateLabel function directly. GetCertificateLabel is called by EAServer when an EAServer client has not specified a certificate label for an SSL connection that requires it.

To override the behavior of any of the functions of the SSLCallBack object, create a standard class user object that descends from SSLCallBack and customize this object as necessary. To let EAServer know which object to use when a callback is required, specify the name of the object in the callbackImpl SSL property. You can set this property value by calling the SetGlobalProperty function.

If you do not provide an implementation of GetCertificateLabel, EAServer receives the CORBA::NO_IMPLEMENT exception and the default implementation of this callback is used. The default implementation always returns the first certificate in the list of labels. If no labels are supplied, the CtsSecurity::NoCertificateException is raised. Any exceptions that may be raised by the function should be added to its prototype.

If your implementation of the callback returns an empty string, the default implementation described above is used and the first certificate label in the list is returned. If the server requires mutual authentication and that certificate is acceptable to the server, the connection proceeds. If the certificate is not acceptable, the connection is refused.

To obtain a useful return value, provide the user with available certificate labels from the labels array passed to the function and ask the user to select one of them. You can also supply additional information obtained from the passed thesessioninfo object.

You can enable the user to cancel the attempt to connect by throwing an exception in this callback function. All exceptions thrown in SSLCallback functions return a CTSSecurity::UserAbortedException to the server. You need to catch the exception by wrapping the ConnectToServer function in a try-catch block.

Examples
This example checks whether any certificate labels are available. To give the user more context, it displays host and port information obtained from the SSL session information object in the message box that informs the user that no certificates are available. If certificates are available, it opens a response window that displays available certificate labels.

The response window returns the text of the selected item using CloseWithReturn:

```c
int idx, numLabels
```
long rc
String ls_rc, sText, sLocation
w_response w_ssl_response
CTSSecurity_sslSessionInfo mySessionInfo

rc = thesessioninfo._narrow(mySessionInfo, &"SessionInfo")
sLocation = mySessionInfo.getProperty("host") + &"
" + mySessionInfo.getProperty("port")
numLabels = upperbound(labels)

IF numLabels <= 0 THEN
    MessageBox("Personal certificate required", &
    "A certificate is required for connection to " &
    + sLocation + "-No certificates are available")
ls_rc = ""
ELSE
    sText = "Available certificates: "
    FOR idx=1 to numLabels
        sText += "-Certificate[" + &
            string(idx) + "]: " + labels[idx]
    NEXT
    OpenWithParm(w_ssl_response, sText)
    ls_rc = Message.StringParm
    IF ls_rc = "cancel" then
        userabortedexception uae
        uae = create userabortedexception
        uae.setmessage("User cancelled connection" &
            + " when asked for certificate")
        throw uae
    END IF
END IF
RETURN ls_rc

See also
ConnectToServer
GetCredentialAttribute
GetPin
TrustVerify

GetChildrenList

Description
Provides a list of the children of a routine included in a trace tree model.
GetChildrenList

Applies to TraceTreeObject, TraceTreeRoutine, and TraceTreeGarbageCollect objects

Syntax

\texttt{instancename.GetChildrenList(list)}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{instancename}</td>
<td>Instance name of the TraceTreeObject, TraceTreeRoutine, or TraceTreeGarbageCollect object.</td>
</tr>
<tr>
<td>\texttt{list}</td>
<td>An unbounded array variable of datatype TraceTreeNode in which GetChildrenList stores a TraceTreeNode object for each child of a routine. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

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:

\begin{verbatim}
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)
  ...
\end{verbatim}
See also BuildModel

**GetColumn**

**Description**
Retrieves column information for a DataWindow, child DataWindow, or ListView control.

For syntax for a DataWindow or a child DataWindow, see the GetColumn method for DataWindows in the *DataWindow Reference* or the online Help.

**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.
alignment | A value of the enumerated datatype 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:

```powerscript
string ls_label, ls_align
int li_width
alignment la_align
```
GetCommandDDE

Description
Obtains the command sent by the client application when your application is a DDE server.

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.

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
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:

``` Powerscript
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:

``` Powerscript
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.

Syntax  

```
GetCommandDDEOrigin ( applstring )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applstring</td>
<td>A string variable in which GetCommandDDEOrigin will store the name of the server application</td>
</tr>
</tbody>
</table>

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.
GetCommandString

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.

This script uses the local variable ls_name to store the name the client application used to identify the server application:

```plaintext
string ls_name
GetCommandDDEOrigin(ls_name)
```

see also

GetCommandDDE
StartServerDDE
StopServerDDE

GetCommandString

Returns the command string sent by dbmlsync to the synchronization server.

Applies to

MLSync controls

Syntax

```plaintext
syncObject.GetCommandString()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syncObject</td>
<td>The name of the MLSync object that starts a synchronization for which you want to get the actual dbmlsync command submitted to the synchronization server.</td>
</tr>
</tbody>
</table>

Return value

String. Returns the command string that is set for submission to the synchronization server. Returns -1 if required properties are missing. (Datasource, Publication, and MLUser, and MLServerVersion are required properties.) When this occurs, a descriptive error is written to the ErrorText property of the MLSync object.

Usage

To minimize confusion to an end user, you might want to hide certain command line arguments in a customized synchronization options window for your MobiLink application. In this case you can call the GetCommandString function to return the command line generated by the options window, then add on values for the options that you hide from the user.

Examples

For MLSync objects, you can allow a user to edit the command line arguments for a synchronization call as follows:

```plaintext
long rc
string cmd
cmd = myMLSync.GetCommandString()
```
GetCompanyName

Description
Returns the company name for the current execution context.

Applies to
ContextInformation objects

Syntax
`servicereference.GetCompanyName ( name )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicereference</td>
<td>Reference to the ContextInformation service instance.</td>
</tr>
<tr>
<td>name</td>
<td>String into which the function places the company name. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

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:

```PowerScript
String ls_company
Integer li_return
ContextInformation ci
ci = create ContextInformation  // or GetContextService("ContextInformation", ci)
li_return = ci.GetCompanyName(ls_company)
IF li_return = 1 THEN
    sle_co_name.text = ls_company
END IF
```

See also
GetContextService
GetFixesVersion
GetHostObject
GetMajorVersion
GetMinorVersion
GetName
GetContextKeywords

Description
Retrieves one or more values associated with a specified keyword.

Applies to
ContextKeyword objects

Syntax
```
 servicereference.GetContextKeywords( name, values )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicereference</td>
<td>Reference to the ContextKeyword service instance.</td>
</tr>
<tr>
<td>name</td>
<td>String specifying the keyword for which the function returns corresponding values.</td>
</tr>
<tr>
<td>values</td>
<td>Unbounded String array into which the function places the values that correspond to name. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

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.

This function can also be used with a PowerBuilder EAServer component to return the value of a specific property from the component’s .props file. To enumerate the properties of an EAServer component, use the Jaguar::Repository API.

Examples
This example calls the GetContextKeywords function:

```powershell
String ls_keyword
Integer li_count, li_return
ContextKeyword lcx_key

li_return = this.GetContextService &
    "ContextKeyword", lcx_key
ls_keyword = s1e_name.Text
lcx_key.GetContextKeywords &
    (ls_keyword, is_values)
FOR li_count = 1 to UpperBound(is_values)
    lbparms.AddItem(is_values[li_count])
NEXT
```

See also
GetContextService
GetContextService

Description
Returns a reference to a context-specific instance of the specified service.

Applies to
Any object

Syntax
GetContextService ( servicename, servicereference )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicename</td>
<td>String specifying the service object. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>• ContextInformation – Context information service</td>
</tr>
<tr>
<td></td>
<td>• ContextKeyword or Keyword – Context keyword service</td>
</tr>
<tr>
<td></td>
<td>(use Keyword as the servicename on all EAServer platforms except AIX)</td>
</tr>
<tr>
<td></td>
<td>• CORBACurrent – CORBA current service for client- or component-management of EAServer transactions</td>
</tr>
<tr>
<td></td>
<td>• ErrorLogging – Error logging service for PowerBuilder components running in a transaction server such as EAServer or COM+</td>
</tr>
<tr>
<td></td>
<td>• Internet – Internet service</td>
</tr>
<tr>
<td></td>
<td>• SSLServiceProvider – SSL service provider service that allows PowerBuilder clients to establish SSL connections to EAServer components</td>
</tr>
<tr>
<td></td>
<td>• TransactionServer – Transaction server service for PowerBuilder components running in a transaction server such as EAServer or COM+</td>
</tr>
<tr>
<td>servicereference</td>
<td>PowerObject into which the function places a reference to the service object specified by servicename. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if the function succeeds and a negative integer if an error occurs. The return value -1 indicates an unspecified error.

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.

In Windows applications, you can use ContextKeyword or Keyword as the servicename argument when you get an instance of the ContextKeyword service. On all EAServer platforms except AIX, you must use Keyword.
GetCredentialAttribute

Using a CREATE statement
You can instantiate these objects with a PowerScript CREATE statement. However, 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 and displays the class of the service in a single line edit box:

```powerbuilder
    Integer li_return
    ContextKeyword lcx_key
    li_return = this.GetContextService &
                 ("Keyword", lcx_key)
    sle_classname.Text = ClassName(lcx_key)
```

See also
BeginTransaction
GetCompanyName
GetContextKeywords
GetHostObject
GetMajorVersion
GetMinorVersion
GetName
GetShortName
GetURL
GetVersionName
HyperLinkToURL
Init
PostURL

GetCredentialAttribute

Description
Called by EAServer to allow the user to supply user credentials dynamically. This function is used by PowerBuilder clients connecting to EAServer.

Applies to
SSLCallBack objects

Syntax
```powerbuilder
sslcallback.GetCredentialAttribute ( thesessioninfo, attr, attrvalues )
```
CHAPTER 10 PowerScript Functions

Return value

String. Returns the selected attribute value.

Usage

A PowerBuilder application does not usually call the GetCredentialAttribute function directly. GetCredentialAttribute is called by EAServer if the useEntrustID property has been set and the EAServer client has not specified the path name of an Entrust INI file or profile.

To override the behavior of any of the functions of the SSLCallBack object, create a standard class user object that descends from SSLCallBack and customize this object as necessary. To let EAServer know which object to use when a callback is required, specify the name of the object in the callbackImpl SSL property. You can set this property value by calling the SetGlobalProperty function.

If you do not provide an implementation of GetCredentialAttribute, EAServer receives the CORBA::NO_IMPLEMENT exception and the default implementation of this callback is used. The default implementation always returns the first value in the list of values supplied. If there are no values supplied, it raises CtsSecurity::NoValueException. Any exceptions that may be raised by the function should be added to its prototype.

If your implementation of the callback returns an empty string, the default implementation described above is used and the first value in the list is returned. If that value is acceptable to the server, the connection proceeds. If the value is not acceptable, the connection is refused.

To obtain a useful return value, provide the user with available attribute values from the attrvalues array passed to the function and ask the user to select one of them. You can also supply additional information, such as the server certificate, obtained from the passed thesessioninfo object.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sslcallback</td>
<td>An instance of a customized SSLCallBack object.</td>
</tr>
<tr>
<td>thesessioninfo</td>
<td>A CORBAObject that contains information about the SSL session. This information can optionally be displayed to the user to provide details about the session.</td>
</tr>
<tr>
<td>attr</td>
<td>A long indicating whether the user needs to specify the path name of an INI file or a profile file. Values are:</td>
</tr>
<tr>
<td></td>
<td>• 1 CRED_ATTR_ENTRUST_INIFILE</td>
</tr>
<tr>
<td></td>
<td>• 2 CRED_ATTR_ENTRUST_USERPROFILE</td>
</tr>
<tr>
<td>attrvalues</td>
<td>An array of string values that contains the available attribute values.</td>
</tr>
</tbody>
</table>
You can enable the user to cancel the attempt to connect by throwing an exception in this callback function. All exceptions thrown in SSLCallback functions return a CTSSecurity::UserAbortedException to the server. You need to catch the exception by wrapping the ConnectToServer function in a try-catch block.

**Examples**

This example checks whether the server requires the location of an INI file or an Entrust user profile and displays an appropriate message. If the `attrvalues` array provides a list of choices, it displays the choices in a message box and prompts the user to enter a selection in a text box:

```plaintext
int idx, numAttrs
String sText, sLocation
numAttrs = upperbound(attrValues)

w_response w_ssl_response

IF attr = 1 THEN
  MessageBox("Entrust INI file required", &
    "Please specify the location of the INI file")
ELSEIF attr = 2 THEN
  MessageBox("Entrust profile required", &
    "Please specify the location of the profile")
END IF

IF numAttrs <> 0 THEN
  sText = "Locations available: ">
  FOR idx = 1 to numAttrs
    sText += "\nattrValues[" + string(idx) + "]: " &
      + attrvalues[idx]
  NEXT
  OpenWithParm( w_ssl_response, SText )
  ls_rc = Message.StringParm
  IF ls_rc = "cancel" then
    userabortedexception uae
    uae = create userabortedexception
    uae.setmessage("User cancelled connection")
    throw uae
  END IF
END IF
RETURN ls_rc
```

See also

ConnectToServer
GetCertificateLabel
GetPin
TrustVerify
CHAPTER 10  PowerScript Functions

GetCurrentDirectory

Description
Gets the current directory for your target application.

Syntax
GetCurrentDirectory ( )

Return value
String. Returns the full path name for the current directory.

Examples
This example puts the current directory name in a SingleLineEdit control:

sle_1.text = GetCurrentDirectory ( )

See also
ChangeDirectory
CreateDirectory
DirectoryExists
RemoveDirectory

GetData

Obtains data from a control.

<table>
<thead>
<tr>
<th>To obtain</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>The value of a data point in a series in a graph</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>The unformatted data from an EditMask control</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Data from an OLE server</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

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

Syntax
controlname.GetData ( { graphcontrol, } seriesnumber, datapoint
{, datatype } )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph from which you want data, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph from which you want the data when controlname is a DataWindow. (DataWindow control only)</td>
</tr>
</tbody>
</table>
**GetData**

**Argument** | **Description**
--- | ---
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 datatype 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:

```powerbuilder
double data_value
integer SeriesNbr
// 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*:

```powerbuilder
double data_value
integer SeriesNbr, ItemNbr
grObjectType MouseHit
MouseHit = &
   gr_prod_data.ObjectAtPointer(SeriesNbr, ItemNbr)
IF MouseHit = TypeSeries! THEN
   data_value = &
      gr_prod_data.GetData(SeriesNbr, ItemNbr)
END IF
```

PowerBuilder
These statements obtain the x value of the data point in the scatter graph `gr_sales_yr` and store it in `data_value`:

```powerscript
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**
```powerscript
editmaskname.GetData ( datavvariable )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>editmaskname</code></td>
<td>The name of the EditMask control containing the data.</td>
</tr>
<tr>
<td><code>datavvariable</code></td>
<td>A variable to which <code>GetData</code> will assign the unformatted data in the EditMask control. The datatype of <code>datavvariable</code> must match the datatype of the EditMask control, which you select in the Window painter. Available datatypes are date, DateTime, decimal, double, string, and time.</td>
</tr>
</tbody>
</table>

**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 datatype of an EditMask control by looking at its MaskDataType property, which holds a value of the MaskDataType enumerated datatype.

**Examples**
- This example gets data of datatype 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`:

```powerscript
date d
em_date.GetData(d)
sle_date.Text = String(d, "mm-dd-yyyy")
```
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:

```powershell
string s
em_string.GetData(s)
sle_string.Text = s
```

### Syntax 3

**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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olename</code></td>
<td>The name of the OLE or custom control containing the object you want to populate with data</td>
</tr>
</tbody>
</table>
| `clipboardformat` | The format for the data. You can specify a standard format with a value of the ClipboardFormat enumerated datatype. 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 0 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 does not 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`:

```powerbuilder
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:

```powerbuilder
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.

**Syntax**
```
GetDataDDE ( string )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string variable in which GetDataDDE will put the data received from a remote DDE application</td>
</tr>
</tbody>
</table>

**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.
CHAPTER 10  PowerScript Functions

GetDataDDEOrigin (applstring, topicstring, itemstring)

Syntax

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applstring</td>
<td>A string variable in which GetDataDDEOrigin will store the name of the server application</td>
</tr>
<tr>
<td>topicstring</td>
<td>A string variable in which GetDataDDEOrigin will store the topic (for example, in Microsoft Excel, the topic could be \textit{REGION.XLS})</td>
</tr>
<tr>
<td>itemstring</td>
<td>A string variable in which GetDataDDEOrigin will store the item identification (for example, in Microsoft Excel, the item could be \textit{R1C2})</td>
</tr>
</tbody>
</table>

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:

```powerscript
string WhichAppl, WhatTopic, WhatLoc
GetDataDDEOrigin(WhichAppl, WhatTopic, WhatLoc)
```

See also

GetDataDDE
OpenChannel
StartServerDDE
StopServerDDE
**GetDataLabelling**

**Description**
Determines whether the data at a given data point is labeled in a DirectX 3D graph.

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

**Syntax**
```
controlname.GetDataLabelling ( (graphcontrol,) series, datapoint, value)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph from which you want data, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control.</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number that identifies the series for which you want the data label setting.</td>
</tr>
<tr>
<td>datapoint</td>
<td>The data point for which you want to obtain a label.</td>
</tr>
<tr>
<td>value</td>
<td>A boolean passed by reference that indicates whether the data point has a label.</td>
</tr>
</tbody>
</table>

**Return value**
Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, GetDataLabelling returns null.

**Usage**
GetDataLabelling determines whether a data label is set for data points from DirectX 3D Area, Bar, Col, or Line graphs. You cannot use this method with DirectX 3D Pie graphs.

**Examples**
In a DataWindow Clicked event, these statements obtain the number of the series and data point clicked by the user in gr_1 and determine whether the label is set for that data point.

```powerbuilder
integer SeriesNbr, ItemNbr
boolean refB
grObjectType clickedtype

clickedtype = this.ObjectAtPointer("gr_1", &SeriesNbr, ItemNbr)

this.GetDataLabelling("gr_1", SeriesNbr, &ItemNbr, refB)
```

These statements obtain the number of the series and data point clicked by the user in a graph object and determine whether the label is set for that data point.

```powerbuilder
integer SeriesNbr, ItemNbr
boolean refB
```
CHAPTER 10  PowerScript Functions

grObjectType clickedtype

clickedtype = this.ObjectAtPointer(SeriesNbr, ItemNbr)

this.GetDataLabelling(SeriesNbr, ItemNbr, refB)

See also  GetSeriesLabelling
         SetDataLabelling
         SetSeriesLabelling

GetDataPieExplode

Description  Reports the percentage of the pie graph’s radius that a pie slice is exploded. An exploded slice is moved away from the center of the pie in order to draw attention to the data.

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

Syntax  controlname.GetDataPieExplode( { graphcontrol, } series, datapoint, percentage )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph for which you want the percentage a pie slice is exploded, or the name of the DataWindow control containing the graph</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control for which you want the percentage a pie slice is exploded</td>
</tr>
<tr>
<td>series</td>
<td>The number that identifies the series</td>
</tr>
<tr>
<td>datapoint</td>
<td>The number of the exploded data point (that is, the pie slice)</td>
</tr>
<tr>
<td>percentage</td>
<td>An integer variable in which you want to store the percentage of the graph’s radius that the pie slice is exploded</td>
</tr>
</tbody>
</table>

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:
GetDataStyle

integer series, datapoint
gObject 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.

<table>
<thead>
<tr>
<th>To get the</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data point’s colors</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Line style and width used by the data point</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Fill pattern or symbol for the data point</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

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 do not 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 the enumerated datatype values that GetDataStyle stores 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

#### Syntax

```
controlname.GetDataStyle( { graphcontrol, } seriesnumber, datapointnumber, colortype, colorvariable )
```

#### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph for which you want the color of a data point, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) When controlname is a DataWindow control, the name of the graph for which you want the color of a data point. (Data Window control only)</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number of the series in which you want the color of a data point.</td>
</tr>
<tr>
<td>datapointnumber</td>
<td>The number of the data point for which you want the color.</td>
</tr>
<tr>
<td>colortype</td>
<td>A value of the grColorType enumerated datatype specifying the aspect of the data point for which you want the color. Values are:</td>
</tr>
<tr>
<td></td>
<td>• Background! – The background color</td>
</tr>
<tr>
<td></td>
<td>• Foreground! – Text (fill color)</td>
</tr>
<tr>
<td></td>
<td>• LineColor! – The color of the line</td>
</tr>
<tr>
<td></td>
<td>• Shade! – The shaded area of three-dimensional graphics</td>
</tr>
<tr>
<td>colorvariable</td>
<td>A long variable in which you want to store the color.</td>
</tr>
</tbody>
</table>

#### 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.

```powerscript
long color_nbr
integer SeriesNbr

// Get the number of the series
SeriesNbr = gr_emp_data.FindSeries("Salary")
```
GetDataStyle

// 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

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

Syntax

controlname.GetDataStyle ( { graphcontrol, } seriesnumber, datapointnumber, linestyle, linewidth )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph for which you want the line style and width of a data point, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph (in the DataWindow control only) for which you want the line style and width of a data point.</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number of the series in which you want the line style and width of a data point.</td>
</tr>
<tr>
<td>datapointnumber</td>
<td>The number of the data point for which you want the line style and width.</td>
</tr>
</tbody>
</table>
CHAPTER 10 PowerScript Functions

**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 \texttt{linestyle} and the line’s width in \texttt{linewidth}. If any argument’s value is \texttt{null}, \texttt{GetDataStyle} returns \texttt{null}.

**Usage**
For the enumerated datatype values that \texttt{GetDataStyle} will store in \texttt{linestyle}, see \texttt{SetDataStyle}.

**Examples**
This example gets the line style and width of data point 10 in the series named Costs in the graph \texttt{gr_product_data}. It stores the information in the variables \texttt{line_style} and \texttt{line_width}:

```powerscript
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 \texttt{sle_series} in the graph \texttt{gr_depts} in the DataWindow control \texttt{dw_employees}. The information is stored in the variables \texttt{line_style} and \texttt{line_width}:

```powerscript
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**
\texttt{FindSeries}  
\texttt{GetSeriesStyle}  
\texttt{SetDataStyle}  
\texttt{SetSeriesStyle}
**GetDataStyle**

### 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
- **Syntax**: 
  
  ```
  controlname.GetDataStyle( { graphcontrol, } seriesnumber, datapointnumber, enumvariable )
  ```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the graph for which you want the fill pattern or symbol type of a data point, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td><code>graphcontrol</code></td>
<td>A string whose value is the name of the graph (in the DataWindow control) for which you want the fill pattern or symbol type of a data point.</td>
</tr>
<tr>
<td><code>seriesnumber</code></td>
<td>The number of the series in which you want the fill pattern or symbol type of a data point.</td>
</tr>
<tr>
<td><code>datapointnumber</code></td>
<td>The number of the data point for which you want the fill pattern or symbol type.</td>
</tr>
<tr>
<td><code>enumvariable</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

- **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 datatype 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 datatype 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)
  ```

- **536**
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:

```power-script
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:

```power-script
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:

```power-script
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
GetDataTransparency

Description
Obtains the transparency percentage of a data point in a DirectX 3D graph (those with 3D rendering).

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

Syntax
controlname.GetDataTransparency ( { graphcontrol, } seriesnumber, datapoint, transparency )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph from which you want data, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control from which you want the data. (DataWindow control only)</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number that identifies the series from which you want data.</td>
</tr>
<tr>
<td>datapoint</td>
<td>The number of the data point for which you want the transparency value.</td>
</tr>
<tr>
<td>transparency</td>
<td>Integer value for percent transparency. A value of 0 means that the data point is opaque and a value of 100 means that it is completely transparent.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, GetDataTransparency returns null.

Usage
GetDataTransparency retrieves data from any DirectX 3D graph (those with 3D rendering).

Examples
These statements obtain the transparency percentage of data point 3 in the series named Costs in the graph gr_computers in the DataWindow control dw_equipment:

```powerbuilder
integer SeriesNbr, rtn, transp_value

// Get the number of the series.
SeriesNbr = dw_equipment.FindSeries( &
   "gr_computers", "Costs")

rtn = dw_equipment.GetDataTransparency( &
   "gr_computers", SeriesNbr, 3, transp_value)
```

These statements obtain the transparency percentage of data point 2 in the series Costs in the graph gr_computers:

```powerbuilder
integer SeriesNbr, rtn, transp_value
```
SeriesNbr = gr_computers.FindSeries("Costs")
rtnc = gr_computers.GetDataTransparency( SeriesNbr, &
  2, transp_value)

See also
FindSeries
GetSeriesTransparency
SetSeriesTransparency
SetDataTransparency

GetDataValue

Description
Obtains the value of a data point in a series in a graph.

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

Syntax
controlname.GetDataValue ( { graphcontrol, } seriesnumber, datapoint,
datavariable {, xory } )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph from which you want data, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control from which you want the data. (DataWindow control only)</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number that identifies the series from which you want data.</td>
</tr>
<tr>
<td>datapoint</td>
<td>The number of the data point for which you want the value.</td>
</tr>
<tr>
<td>datavariable</td>
<td>The name of a variable that will hold the data value. The variable’s datatype can be date, DateTime, double, string, or time. The variable must have the same datatype as the values axis of the graph.</td>
</tr>
<tr>
<td>xory</td>
<td>(Optional) A value of the grDataType enumerated datatype 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 (scatter graph only)</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, GetDataValue returns null.
**GetDataValue**

**Usage**

GetDataValue retrieves data from any graph. The data is stored in *datavariable*, whose datatype must match the datatype 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`:

```plaintext
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 does not click on a data point, then `ItemNbr` is set to 0. The categories of the graph are time values:

```plaintext
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 does not click on a data point, then `ItemNbr` is set to 0. The datatype of the category axis is `Date`:

```plaintext
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
```
GetDateLimits

Description
Retrieves the maximum and minimum date limits specified for the calendar.

Applies to
MonthCalendar control

Syntax
controlname.GetDateLimits( min, max )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the MonthCalendar control for which you want to determine the date limits</td>
</tr>
<tr>
<td>min</td>
<td>A date value returned by reference that represents the minimum date that can be referenced or displayed in the calendar</td>
</tr>
<tr>
<td>max</td>
<td>A date value returned by reference that represents the maximum date that can be referenced or displayed in the calendar</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 0 when both limits are retrieved successfully and one of the following negative values otherwise:

-1  No limits were set
-2  Unknown failure

Usage
Use the SetDateLimits function to set minimum and maximum dates. If no date limits have been set, GetDateLimits returns -1 and sets min and max to January 1, 1900.

Examples
This example displays a message box that shows the minimum and maximum dates set for a control:

```powerShell
integer li_return
Date mindate, maxdate
string str1, str2

li_return = mc_1.GetDateLimits(mindate, maxdate)
If li_return = -1 then
    str1 = "No minimum and maximum dates are set"
elseif li_return = -2 then
    str1 = "Unknown failure"
else
```

See also
DeleteData
FindSeries
InsertData
ObjectAtPointer
GetDbmsyncPath

str1 = "Minimum date is " + string(mindate)
str2 = "Maximum date is " + string(maxdate)
end if

MessageBox("Date Limits", str1 + "-r-n" + str2)

See also
SetDateLimits

GetDbmsyncPath

Description
Retrieves the full path and file name of the dbmsync.exe that is installed on the workstation.

Applies to
MLSync and MLSynchronization controls

Syntax
SyncObject.GetDbmsyncPath()

Argument | Description
--- | ---
syncObject | The name of the synchronization object

Return value
String. Returns the value of full path and file name of the synchronization executable. Returns -1 if dbmsync.exe is not found.

Usage
This property uses the value of the synchronization object’s MLServerVersion property to decide which version of dbmsync to return.

Examples
This function enables the use of Help buttons for the Additional Options and Extended Options fields as displayed on the default ML Server tab page of the w_appname_sync_options dialog box that is generated by the MobiLink synchronization wizard. The Clicked event for these Help buttons has the following script:

```
string sCmd
sCmd = i_uosync.GetDbmsyncPath()
if len(sCmd) > 0 then
    sCmd += " ?"
    // sCmd += ' -l' for help with extended options
    run(sCmd, Normal!)
end if
```
GetDisplayRange

Description
Retrieves the first and last date of the currently displayed date range and returns the number of months that span the display.

Applies to
MonthCalendar control

Syntax
`controlname.GetDisplayRange ( start, end {, d } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the MonthCalendar control for which you want the range of dates</td>
</tr>
<tr>
<td><code>start</code></td>
<td>A date specifying the first date in the displayed range returned by reference</td>
</tr>
<tr>
<td><code>end</code></td>
<td>A date specifying the last date in the displayed range returned by reference</td>
</tr>
</tbody>
</table>
| `d` (optional) | A value of the MonthCalDisplayState enumerated variable. Values are:  
EntirelyDisplayed! – Gets the range of dates for which all days in each month are displayed  
PartlyDisplayed! – Gets the range of dates for which any days in each month are displayed (default) |

Return value
Integer. Returns the number of months in the display range if it succeeds and -1 if it fails.

Usage
The GetDisplayRange function retrieves the beginning and end dates of the range of dates currently displayed in the calendar.

If you do not supply the optional `d` argument (or specify PartlyDisplayed!), GetDisplayRange returns the number of months for which any of the days in the month display. If the calendar displays one month, the return value is 3, because the last few days of the previous month and the first few days of the next month are included.

If you supply EntirelyDisplayed! as the `d` argument, GetDisplayRange returns the number of months for which all of the days in the month display. It ignores the leading and trailing days.

For example, if the calendar display shows the 12 months from November 2004 to October 2005 and you do not supply the `d` argument, GetDisplayRange returns 14 and the `start` and `end` arguments are set to October 25, 2004 and November 6, 2005.

If you supply EntirelyDisplayed! as the `d` argument, GetDisplayRange returns 12 and the `start` and `end` arguments are set to November 1, 2004 and October 31, 2005.
This example displays a message box that shows the number of months in the display range and its start and end dates. Because the third argument is set to `PartlyDisplayed!`, the range returned will be greater than the number of full months displayed. If only one month displays and it neither begins on the first day of the week nor ends on the last day of the week, `li_return` will be 3:

```powershell
integer li_return
Date startdate, enddate
string str1, string str2

li_return = mc_1.GetDisplayRange(startdate, enddate, & PartlyDisplayed!)
str1 = "Range is " + string(li_return) + " months"
str2 = "Start date is " + string(startdate) + "~r~n"
str2 += "End date is " + string(enddate)

MessageBox(str1, str2)
```

This example finds out how many complete months are shown in the current display and sets the scroll rate to that number:

```powershell
integer li_return
Date startdate, enddate

li_return = mc_1.GetDisplayRange(startdate, enddate, & EntirelyDisplayed!)
mc_1.ScrollRate = li_return
```

See also

GetSelectedRange

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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicDescriptionArea</td>
<td>The name of the DynamicDescriptionArea, usually SQLDA.</td>
</tr>
<tr>
<td>index</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 datatype specifying the datatype 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:

```powerScript
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 datatype 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:

```powerScript
Date Datevar
Time Timevar
DateTime Datetimevar
Double Doublevar
String Stringvar
```
GetDynamicDateTime

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!
            Datetimexvar = SQLDA.GetDynamicDateTime(n)
            ... // Other processing
        CASE TypeTime!
            Timevar = SQLDA.GetDynamicTime(n)
            ... // Other processing
        CASE ELSE
            MessageBox("Dynamic SQL", &
                        "datatype 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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicDescriptionArea</td>
<td>The name of the DynamicDescriptionArea, usually SQLDA.</td>
</tr>
<tr>
<td>index</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 SystemDateTime to the DateTime data in the second output parameter descriptor:

```powerscript
DateTime SystemDateTime
SystemDateTime = SQLDA.GetDynamicDateTime(2)
IF SystemDateTime = &Date(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
GetDynamicDecimal

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

\[
\text{DynamicDescriptionArea.GetDynamicDecimal}( \text{index} )
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicDescriptionArea</td>
<td>The name of the DynamicDescriptionArea, usually SQLDA.</td>
</tr>
<tr>
<td>index</td>
<td>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.</td>
</tr>
</tbody>
</table>

Return value
Decimal. 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, GetDynamicDecimal returns null.

Usage
Use GetDynamicDecimal when the value of OutParmType is TypeDecimal! or TypeLongLong! 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:

```plaintext
Integer DeptId
DeptId = SQLDA.GetDynamicDecimal(2)
```

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)
```

**Return value**
Double. 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 TypeByte!, TypeInteger!, TypeDouble!, TypeLong!, TypeReal!, or TypeBoolean! for the value that you want to retrieve from the array.

For OutParmType values of TypeDecimal! or TypeLongLong!, use GetDynamicDecimal instead.

**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
GetDynamicDecimal
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:

```powerbuilder
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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicDescriptionArea</td>
<td>The name of the DynamicDescriptionArea, usually SQLDA.</td>
</tr>
<tr>
<td>index</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```powerShell
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. 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:

```powerbuilder
string path
environment env
integer rtn

rtn = GetEnvironment(env)
IF rtn <> 1 THEN RETURN

CHOOSE CASE env.OSType
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)
```
This example displays a message box that shows the major, minor, and fixes versions and the build number of PowerBuilder:

```powerscript
string ls_version
environment env
integer rtn

rtn = GetEnvironment(env)

IF rtn <> 1 THEN RETURN
ls_version = "Version: " + string(env.pbmajorrevision)
ls_version += "." + string(env.pbminorrevision)
ls_version += "." + string(env.pbfixesrevision)
ls_version += " Build: " + string(env.pbbuildnumber)

MessageBox("PowerBuilder Version", ls_version)
```

### GetFileOpenName

**Description**
Displays the system’s Open File dialog box and allows the user to select a file or enter a file name.

**Syntax**

```powerscript
GetFileOpenName ( title, pathname, filename {, extension {, filter {, initdir {, aFlag } } } } )
GetFileOpenName ( title, pathname, filename[ ] {, extension {, filter {, initdir {, aFlag } } } } )
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>A string whose value is the title of the dialog box.</td>
</tr>
<tr>
<td>pathname</td>
<td>A string variable in which you want to store the returned path. If the user selects a single file, the <code>pathname</code> variable contains the path name and file name.</td>
</tr>
<tr>
<td>filename, filename[ ]</td>
<td>A string variable in which the returned file name is stored or an array of string variables in which multiple selected file names are stored. Specifying an array of string variables enables multiple selection in the dialog box.</td>
</tr>
<tr>
<td>extension (optional)</td>
<td>A string whose value is a 1- to 3-character default file extension. The default is no extension.</td>
</tr>
</tbody>
</table>
GetFileOpenName

**Argument** | **Description**
--- | ---
*filter* (optional) | A string whose value is a text description of the files to include in the list box 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
```

To specify multiple filter patterns for a single display string, use a semicolon to separate the patterns, for example:

```
"Graphic Files (*.bmp;*.gif;*.jpg;*.jpeg),
*.bmp;*.gif;*.jpg;*.jpeg"
```

The default is:

```
"All Files (*.*)",".*"
```

*initdir* (optional) | A string whose value is the initial directory name. The default is the current directory.

*aFlag* (optional) | An unsigned long whose value determines which options are enabled in the dialog box. The value of each option’s flag is calculated as 2 to the power of (index - 1), where index is the integer associated with the option. The value of the aggregate flag passed to GetFileOpenName is the sum of the individual option flags. See the table in the Usage section for a list of options, the index associated with each option, and the option’s meaning.

**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**

If you specify a DOS-style file extension and the user enters a file name with no extension, PowerBuilder appends the default extension to the file name. If you specify a file mask to act as a filter, PowerBuilder displays only files that match the mask.

If you specify a string for the *filename* argument, the user can select only one file. The *pathname* argument contains the path name and the file name, for example `C:\temp\test.txt`.

If you specify a string array for the *filename* argument, the user can select more than one file. If the user selects multiple files, the *pathname* argument contains the path only, for example `C:\temp`. If the user selects a single file, its name is appended to the *pathname* argument, for example `C:\temp\test.txt`.

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"

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, unless the optional initdir argument is passed.

The aFlag argument is used to pass one or more options that determine the appearance of the dialog box. For each option, the value of the flag is \(2^{(index - 1)}\), where index is an integer associated with each option as shown in the following table. You can pass multiple options by passing an aggregate flag, calculated by adding the values of the individual flags.

If you do not pass an aFlag, the Explorer-style open file dialog box is used. If you do pass a flag, the old-style dialog box is used by default. Some options do not apply when the Explorer-style dialog box is used. For those that do apply, add the option value for using the Explorer-style dialog box (2) to the value of the option if you want to display an Explorer-style dialog box.

For example, passing the flag 32768 \(2^{15}\) to the GetFileSaveName function opens the old-style dialog box with the Read Only check box selected by default. Passing the flag 32770 opens the Explorer-style dialog box with the Read Only check box selected by default.

<table>
<thead>
<tr>
<th>Index</th>
<th>Constant name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OFN_CREATEPROMPT</td>
<td>If the specified file does not exist, prompt for permission to create the file. If the user chooses to create the file, the dialog box closes; otherwise the dialog box remains open.</td>
</tr>
<tr>
<td>2</td>
<td>OFN_EXPLORER</td>
<td>Use an Explorer-style dialog box.</td>
</tr>
<tr>
<td>3</td>
<td>OFN_EXTENSIONDIFFERENT</td>
<td>The file extension entered differed from the extensions specified in extension.</td>
</tr>
<tr>
<td>4</td>
<td>OFN_FILEMUSTEXIST</td>
<td>Only the names of existing files can be entered.</td>
</tr>
<tr>
<td>5</td>
<td>OFN_HIDEREADONLY</td>
<td>Hide the Read Only check box.</td>
</tr>
<tr>
<td>6</td>
<td>OFN_LONGNAMES</td>
<td>Use long file names. Ignored for Explorer-style dialog boxes.</td>
</tr>
<tr>
<td>7</td>
<td>OFN_NOCHANGEDIR</td>
<td>Restore the current directory to its original value if the user changed the directory while searching for files. This option has no effect for GetFileOpenName on Windows NT, 2000, and XP.</td>
</tr>
</tbody>
</table>
**Opening a file**

Use the FileOpen function to open a selected file.

**Examples**

The following example displays a Select File dialog box that allows multiple selection. The file types are TXT, DOC, and all files, and the initial directory is `C:\Program Files\Sybase`. The option flag 18 specifies that the Explorer-style dialog box is used `(2^1 = 2)`, and the Read Only check box is hidden `(2^4 = 16)`. The selected filenames are displayed in a MultiLineEdit control.

If the user selects a single file, the `docpath` variable contains both the path and the file name. The example contains an IF clause to allow for this.

```pascal
string docpath, docname[]
integer i, li_cnt, li_rtn, li_filenum
```
li_rtn = GetFileOpenName("Select File", &
docpath, docname[], "DOC", &
+ "Text Files (*.TXT),*.TXT," &
+ "Doc Files (*.DOC),*.DOC," &
+ "All Files (*.*), *.*", &
"C:\Program Files\Sybase", 18)

mle_selected.text = ""
IF li_rtn < 1 THEN return
li_cnt = Upperbound(docname)

// if only one file is picked, docpath contains the
// path and file name
if li_cnt = 1 then
    mle_selected.text = string(docpath)
else

    // if multiple files are picked, docpath contains the
    // path only - concatenate docpath and docname
    for i=1 to li_cnt
        mle_selected.text += string(docpath) &
        + "\"+(string(docname[i]))+"r\n"
    next
end if

In the following example, the dialog box has the title Open and displays text
files, batch files, and INI files in the Files of Type drop-down list. The initial
directory is d:\temp. The option flag 512 specifies that the old-style dialog box
is used and the Network button is hidden (2^9 = 512).

// instance variables
// string is_filename, is_fullname
int li_fileid

if GetFileOpenName("Open", is_fullname, is_filename, &
"txt", "Text Files (*.txt),*.txt,INI Files " &
+ "(*.ini), *.ini,Batch Files (*.bat),*.bat", &
"d:\temp", 512) < 1 then return

li_fileid = FileOpen (is_fullname, StreamMode!)
FileRead (li_fileid, mle_notepad.text)
FileClose (li_fileid)

See also
DirList
DirSelect
GetFileSaveName
GetFolder
GetFileSaveName

Description
Displays the system’s Save File dialog box with the specified file name displayed in the File name box. The user can enter a file name or select a file from the grayed list.

Syntax

\[
\textbf{GetFileSaveName}(\text{title}, \text{pathname}, \text{filename}[, \text{extension}[, \text{filter}[, \text{initdir}[, \text{aFlag}]]]])
\]

\[
\textbf{GetFileSaveName}(\text{title}, \text{pathname}, \text{filename}[, \text{extension}[, \text{filter}[, \text{initdir}[, \text{aFlag}]]]])
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>A string whose value is the title of the dialog box.</td>
</tr>
<tr>
<td>pathname</td>
<td>A string variable whose value is the default path name and which stores the returned path. If the user selects a single file, the pathname variable contains the path name and file name. The default file name is displayed in the File name box; the user can specify another name.</td>
</tr>
<tr>
<td>filename, filename[]</td>
<td>A string variable in which the returned file name is stored or an array of string variables in which multiple selected file names are stored. Specifying an array of string variables enables multiple selection in the dialog box.</td>
</tr>
<tr>
<td>extension (optional)</td>
<td>A string whose value is a 1- to 3-character default file extension. The default is no extension.</td>
</tr>
<tr>
<td>filter (optional)</td>
<td>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.<em>, ext The default is: &quot;All Files (</em>.<em>)&quot;,.</em>&quot;</td>
</tr>
<tr>
<td>initdir (optional)</td>
<td>A string whose value is the initial directory name. The default is the current directory.</td>
</tr>
<tr>
<td>aFlag (optional)</td>
<td>An unsigned long whose value determines which options are enabled in the dialog box. The value of each option’s flag is calculated as 2 to the power of (index -1), where index is the integer associated with the option. The value of the aggregate flag passed to GetFileOpenName is the sum of the individual option flags. See the table in the Usage section for GetFileOpenName for a list of options, the index associated with each option, and the option’s meaning.</td>
</tr>
</tbody>
</table>

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.
CHAPTER 10  PowerScript Functions

Usage

If you specify a DOS-style extension and the user enters a file name with no extension, PowerBuilder appends the default extension to the file name. If you specify a file mask to act as a filter, PowerBuilder displays only files that match the mask.

If you specify a string for the filename argument, the user can select only one file. The pathname argument contains the path name and the file name, for example C:\temp\test.txt.

If you specify a string array for the filename argument, the user can select more than one file. If the user selects multiple files, the pathname argument contains the path only, for example C:\temp. If the user selects a single file, its name is appended to the pathname argument, for example C:\temp\test.txt. For an example that shows the use of a string array, see the GetFileOpenName function.

For usage notes on the filter, initdir, and aFlag arguments, see the GetFileOpenName function.

Examples

These statements display the Select File dialog box so that the user can select a single file. The default file extension is .DOC, the filter is all files, and the initial directory is C:\My Documents. The aFlag option 32770 specifies that an Explorer-style dialog box is used with the Read Only check box selected when the dialog box is created. If a file is selected successfully, its path displays in a SingleLineEdit control:

```
string ls_path, ls_file
int li_rc

ls_path = sle_1.Text
li_rc = GetFileSaveName ( "Select File", &
    ls_path, ls_file, "DOC", &
    "All Files (*.*) , *.*", "C:\My Documents", &
    32770)

IF li_rc = 1 Then
    sle_1.Text = ls_path
End If
```

See also

DirList
DirSelect
GetFileOpenName
GetFolder
GetFirstSheet

Description Obtains the top sheet in the MDI frame, which may or may not be active.

Applies to MDI frame windows

Syntax

\[ \text{mdiframewindow.GetFirstSheet}() \]

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 \text{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 10.2.1, the fix version is 1.

**Applies to**
ContextInformation objects

**Syntax**
```
servicereference.GetFixesVersion ( fixversion )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicereference</td>
<td>Reference to the ContextInformation service instance.</td>
</tr>
<tr>
<td>fixversion</td>
<td>Integer into which the function places the fix version. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

**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:

```powerscript
String ls_name
Constant String ls_currver = "8.0.3"
Integer li_majver, li_minver, li_fixver
ContextInformation ci

this.GetContextService ("ContextInformation", ci)
ci.GetMajorVersion(li_majver)
ci.GetMinorVersion(li_minver)
ci.GetFixesVersion(li_fixver)
IF li_majver <> 8 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 <> 3 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 datatype of the control that currently has focus, and then set text_value to the text property of the control:

```powerbuilder
GraphicObject which_control
SingleLineEdit sle_which
CommandButton cb_which
string text_value

which_control = GetFocus()

CHOICE 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 CHOICE
```

See also
IsValid
SetFocus
GetFolder

Displays a folder selection dialog box.

**Syntax**

```
GetFolder ( title, directory )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>String for a title that displays above a list box containing a tree view for folder selection.</td>
</tr>
<tr>
<td>directory</td>
<td>String for the directory name passed by reference to the folder selection dialog box. The directory name is selected, and its subfolders, if any, are displayed in a dialog box tree view.</td>
</tr>
</tbody>
</table>

**Return value**

Integer. Returns 1 if the function succeeds, 0 if the user selects cancel (or the dialog box is closed), -1 if an error occurs.

**Usage**

The directory selected by the user is returned in the same variable that is passed to the folder selection dialog box.

**Examples**

This example displays the folder contents of the Sybase directory in a folder selection dialog box. The string passed in the `title` argument displays above the tree view:

```plaintext
string ls_path = "d:\program files\sybase"
integer li_result
li_result = GetFolder("my targets", ls_path)
sle_1.text=ls_path
// puts the user-selected path in a SingleLineEdit box.
```

**See also**

DirectoryExists
DirList
DirSelect
GetCurrentDirectory
GetFileOpenName
GetFileSaveName
GetGlobalProperty

Returns the value of an SSL global property. This function is used by PowerBuilder clients connecting to EAServer.

Applies to

SSLServiceProvider object

Syntax

```
sslServiceProvider.GetGlobalProperty ( property, values )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sslServiceProvider</td>
<td>Reference to the SSLServiceProvider service instance.</td>
</tr>
<tr>
<td>property</td>
<td>The name of the SSL property for which you want to return values.</td>
</tr>
<tr>
<td></td>
<td>For a complete list of supported SSL properties, see your EAServer documentation or the online Help for the Connection object.</td>
</tr>
<tr>
<td>values</td>
<td>An array of string values for the specified SSL property.</td>
</tr>
</tbody>
</table>

Return value

Long. Returns one of the following values:

- 0 Success
- -1 Unknown property
- -3 Property has no value
- -10 An EAServer or SSL failure has occurred
- -11 Bad argument list

Usage

The GetGlobalProperty function allows PowerBuilder clients that connect to EAServer through SSL to access global SSL properties.

Any properties set using the SSLServiceProvider interface are global to all connections made by the client to all EAServer servers. You can override any of the global settings at the connection level by specifying them as options to the Connection object or JaguarORB object.

Only clients can get and set SSL properties. Server components do not have permission to use the SSLServiceProvider service.

Examples

The following example shows the use of the GetGlobalProperty function to get the value of the sessLingerTime property:

```
SSLServiceProvider ssl
string ls_values[]
long rc
...
this.GetContextService("SSLServiceProvider", ssl)
rc = ssl.GetGlobalProperty("sessLingerTime", ls_values)
...
```

See also

SetGlobalProperty

564

PowerBuilder
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.

<table>
<thead>
<tr>
<th>Applies to</th>
<th>ContextInformation objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>servicereference.GetHostObject ( hostobject )</td>
</tr>
<tr>
<td>Argument</td>
<td>Description</td>
</tr>
<tr>
<td>servicereference</td>
<td>Reference to the Context Information service instance</td>
</tr>
<tr>
<td>hostobject</td>
<td>PowerObject into which the function places a reference to the ActiveX automation server object</td>
</tr>
</tbody>
</table>

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 using the GetContextService function; ole1 is an instance variable of type OLEObject:

```power_script
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
GetName
GetShortName
GetVersionName
GetItem

Retrieves data associated with a specified item in ListView and TreeView controls.

<table>
<thead>
<tr>
<th>To retrieve data associated with a specified item</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListView control item</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>ListView control item and column</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>TreeView item</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For ListView controls**

Retrieves a ListViewItem object from a ListView control so you can examine its properties.

**Applies to** ListView controls

**Syntax**

```
listviewname.GetItem(index, {column}, item)
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control for which you want to retrieve the ListViewItem item</td>
</tr>
<tr>
<td>index</td>
<td>The index number of the item you want to retrieve</td>
</tr>
<tr>
<td>column</td>
<td>The index number of the column for which you want item information</td>
</tr>
<tr>
<td>item</td>
<td>The ListViewItem variable in which you want to store the ListViewItem object</td>
</tr>
</tbody>
</table>

**Return value**

Integer. Returns 1 if it succeeds and -1 if an error occurs. Stores a ListViewItem object in a ListViewItem variable.

**Usage**

You can retrieve properties for any ListView item with this syntax. If you do not specify a column, GetItem retrieves properties for the first column of an item. Only report views display multiple columns.

To retrieve labels only, use syntax 2. You can use GetColumn to obtain column properties that are not specific to a ListView item.

To change pictures and other property values associated with a ListView item, use GetItem, change the property values, and use SetItem to apply the changes back to the ListView.
Examples

This example uses `GetItem` to move the second item in the `lv_list` ListView control to the fifth item. It retrieves item 2, inserts it into the ListView control as item 5, and then deletes the original item:

```powerscript
listviewitem l_lvi

lv_list.GetItem(2, l_lvi)
lv_list.InsertItem(5, l_lvi)
lv_list.DeleteItem(2)
```

See also

`GetColumn`

`SetItem`

**Syntax 2**

For ListView controls

Description

Retrieves the value displayed for a ListView item in a specified column.

Applies to

ListView controls

Syntax

```
listviewname.GetItem(index, column, label)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listviewname</code></td>
<td>The name of the ListView control from which you want to retrieve a displayed value.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>The index number of the item for which you want to retrieve a displayed value.</td>
</tr>
<tr>
<td><code>column</code></td>
<td>The index number of the column for which you want to retrieve a value. If the ListView is not a multicolumn report view, all the items are considered to be in column 1.</td>
</tr>
<tr>
<td><code>label</code></td>
<td>A string variable in which you store the displayed value.</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. Stores the displayed value of the ListView column in a string variable.

Usage

To retrieve property values for a ListView item, use Syntax 1.

Examples

This example gets the displayed values from column 1 and column 3 of the first row of the `lv_list` ListView and displays them in the `sle_info` SingleLineEdit control.

```powerscript
string ls_artist, ls_comp

lv_list.GetItem(1, 1, ls_comp)
lv_list.GetItem(1, 3, ls_artist)

sle_info.text = ls_artist +": wrote " + ls_comp + "."
```

See also

`SetItem`
**Syntax 3**

**For TreeView controls**

Retrieves the data associated with the specified item.

**Applies to**

TreeView controls

**Syntax**

\[ \text{treeviewname}.\text{.GetItem} (\text{itemhandle}, \text{item}) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The name of the TreeView control in which you want to get data for a specified item</td>
</tr>
<tr>
<td>itemhandle</td>
<td>The handle for the item for which you want to retrieve information</td>
</tr>
<tr>
<td>item</td>
<td>A TreeViewItem variable in which you want to store the item identified by the item handle</td>
</tr>
</tbody>
</table>

**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 item in a TreeView (such as label, handle, or picture index). After you have retrieved the information, you can use it in your application. To change a property of an item, 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:

```powerbuilder
treeviewitem tvi
This.SetItem(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:

```powerbuilder
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
GetItemAtPointer

Description
Gets the handle or the index of the item under the cursor.

Applies to
ListView controls, TreeView controls

Syntax
\[ controlname.GetItemAtPointer() \]

Return value
Long. Returns the index (ListView) or handle (TreeView) of the item under the cursor. Returns -1 for failure.

Usage
System events that select an item in a ListView or TreeView control, such as the Clicked event, already have an argument that passes the index for the ListView or the handle for the TreeView. The GetItemAtPointer function allows you to retrieve the index or handle in user events (or system events without an index or handle argument) for a ListView or TreeView control.

Examples
This example places the handle of a TreeView item in a SingleLineEdit box:

```powerbuilder
integer li_index
li_index = tv_1.GetItemAtPointer()
sle_1.text = string(li_index)
```

See also
FindItem
SelectItem

GetLastReturn

Description
Returns the return value from the last InvokePBFunct or TriggerPBEvent function.

Applies to
Window ActiveX controls

Syntax
\[activexcontrol.GetLastReturn() \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activexcontrol</td>
<td>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.</td>
</tr>
</tbody>
</table>
GetLibraryList

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 using an argument in InvokePBFunction or TriggerPBEvent.

Examples
This JavaScript example calls the GetLastReturn function:

```javascript
... 
	retcd = PBRX1.TriggerPBEvent(theEvent, numargs);
	rc = parseInt(PBRX1.GetLastReturn());

	if (rc != 1) { 
		alert("Error. Empty string.");
	}

... 
```

This VBScript example calls the GetLastReturn function:

```vbscript
... 
	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

GetLibraryList

Description
Gets the files in the library search path of the application.

Syntax
`GetLibraryList()`

Return value
String. Returns the current library list with complete paths. Multiple libraries are separated by commas.

Usage
You should call GetLibraryList and append any libraries you want to add to the list before updating the search path using the SetLibraryList function.
Examples

This example obtains the list of libraries, adds a library to the list, then resets
the list:

```powerbuilder
string ls_list, ls_newlist

ls_list = getlibrarylist ()
ls_newlist = ls_list + ",c:\my_library.pbl"
setlibrarylist (ls_newlist)
```

See also
AddToLibraryList
SetLibraryList

---

### GetMajorVersion

**Description**
Returns the major version for the current PowerBuilder execution context. For
example, at maintenance level 10.2.1 the major version is 10.

**Applies to**
ContextInformation objects

**Syntax**
```
servicereference.GetMajorVersion ( majorversion )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicereference</td>
<td>Reference to the ContextInformation service instance.</td>
</tr>
<tr>
<td>majorversion</td>
<td>Integer into which the function places the major version. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

**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:

```powerbuilder
String ls_name
Constant String ls_currver = "8.0.3"
Integer li_majver, li_minver, li_fixver
ContextInformation ci

this.GetContextService ("ContextInformation", ci)

GetMajorVersion(li_majver)
ci.GetMinorVersion(li_minver)
ci.GetFixesVersion(li_fixver)
IF li_majver <> 8 THEN
    MessageBox("Error", 
        "Must be at Version " + ls_currver)
ELSEIF li_minver <> 0 THEN
```

---
GetMessage

Description
Returns the error message from objects of type Throwable.

Syntax
throwableobject.GetMessage()

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>throwableobject</td>
<td>Object of type Throwable from which you want to retrieve an error message</td>
</tr>
</tbody>
</table>

Return value
String. The error text for system error objects, such as RuntimeException, is preset.

Usage
You can set the error message for an object of type Throwable using the SetMessage function.

Examples
This example catches a system error message and displays that error in a message box. Catching the system error prevents the application from terminating when the arccosine argument, entered by the application user, is not in the required range:

```powerbuilder
Double ld_num
ld_num = Double (sle_1.text)
TRY
  sle_2.text = string (acos (ld_num))
CATCH (runtimeerror er)
    MessageBox("Runtime Error", er.GetMessage())
END TRY
```
This example catches and displays a user error message from the Clicked event of a button that calls the user-defined function, \texttt{wf_acos}. The user-defined function catches a runtime error—preventing the application from terminating—and then sets the message for a user object, \texttt{uo\_exception}, that inherits from the Exception object type:

\begin{verbatim}
TRY
    wf_acos()
CATCH (uo_exception u_ex)
    messagebox("Out of Range", u_ex.GetMessage())
END TRY
\end{verbatim}

Code for the \texttt{wf_acos} function is shown in the \texttt{SetMessage} function.

See also \texttt{SetMessage}

---

### GetMinorVersion

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the minor version for the current PowerBuilder execution context. For example, at maintenance level 10.2.1 the minor version is 2.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies to</td>
<td>ContextInformation objects</td>
</tr>
<tr>
<td>Syntax</td>
<td>\texttt{servicereference.GetMinorVersion ( minorversion )}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicereference</td>
<td>Reference to the ContextInformation service instance.</td>
</tr>
<tr>
<td>minorversion</td>
<td>Integer into which the function places the minor version. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

| 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:                                                                        |

\begin{verbatim}
String ls_name
Constant String ls_currver = "8.0.3"
Integer li_majver, li_minver, li_fixver
ContextInformation ci

    this.GetContextService("ContextInformation", ci)

    ci.GetMajorVersion(li_majver)
    ci.GetMinorVersion(li_minver)
    ci.GetFixesVersion(li_fixver)
\end{verbatim}
IF li_majver <> 8 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 <> 3 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.GetNam( name )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicereference</td>
<td>Reference to the ContextInformation service instance.</td>
</tr>
<tr>
<td>name</td>
<td>String into which the function places the name. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage
Call this function to determine the current execution environment.

The window plug-in and window ActiveX contexts are obsolete in the current version of PowerBuilder. For PowerBuilder 11.5 applications, the only value passed for the name argument is “PowerBuilder Runtime”.

Examples
This example calls the GetName function. ci is an instance variable of type ContextInformation:

    String ls_name
this.getContextService("ContextInformation", ci)
ci.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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olename</td>
<td>The name of the OLE control containing the object for which you want the native pointer.</td>
</tr>
<tr>
<td>pointer</td>
<td>A UnsignedLong variable in which you want to store the pointer. If GetNativePointer cannot get a valid pointer, pointer is set to 0.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage
Pointer is a pointer to OLE’s IUnknown interface. You can use it with the OLE QueryInterface function 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.

Only for external DLL calls
This function is only useful for external DLL calls. It is not related to the SetAutomationPointer function.
GetNextSheet

Examples

This example gets a pointer for the OLECustomControl ocx_spell for making external function calls for OLE automation:

```powershell
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

GetNextSheet

Description

Obtains the sheet that is behind the specified sheet in the MDI frame.

Applies to

MDI frame windows

Syntax

`mdiframewindow.GetNextSheet ( sheet )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mdiframewindow</code></td>
<td>The MDI frame window in which you want the next sheet</td>
</tr>
<tr>
<td><code>sheet</code></td>
<td>The sheet for which you want the sheet that is behind it</td>
</tr>
</tbody>
</table>

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 is not valid.
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:

```power_script
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

---

### GetObjectRevisionFromRegistry

**Description**
Assigns synchronization property values saved in the Windows registry to a synchronization object.

**Applies to**
MLSynchronization, MLSync controls

**Syntax**
```
SyncObject.GetObjectRevisionFromRegistry()
```

**Argument**
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syncObject</td>
<td>The name of the synchronization object</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns the value of ObjectRevision. Returns -1 if the registry key is not found or if the SyncRegistryKey property of the synchronization object is not set.

**Usage**
The registry key is located under the HKEY_CURRENT_USER\SyncRegistryKey\DBSyncType where SyncRegistryKey is the value of the synchronization object’s SyncRegistryKey property and DBSyncType is MobiLink.

**See also**
- GetSyncRegistryProperties
- SetSyncRegistryProperties
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:

```powershell
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**

```powerlanguage
econtrol.GetParagraphSetting ( whichsetting )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>econtrol</code></td>
<td>The name of the control for which you want paragraph information.</td>
</tr>
</tbody>
</table>
| `whichsetting`| A value of the ParagraphSetting enumerated datatype 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:

```powerlanguage
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**

```powerlanguage
objectname.GetParent ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectname</code></td>
<td>A control in a window or user object or an item on a menu for which you want the parent object</td>
</tr>
</tbody>
</table>

**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:

```powershell
// a_tab is generic;
// it does not know about specific pages
Tab a_tab

// a_tab_page is generic;
// it does not 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 cannot refer to controls on the user object because `a_tab_page` does not know about them. You cannot refer to specific pages in the Tab control because `a_tab` does not 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:

```powershell
// 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 11
GetPin

Description
Called by EAServer to obtain a PIN for use with an SSL connection. This function is used by PowerBuilder clients connecting to EAServer.

Applies to
SSLCallback objects

Syntax
sslcallback.GetPin ( thesessioninfo, timedout )

Argument | Description
---|---
sslcallback | An instance of a customized SSLCallback object.
thesessioninfo | A CORBAObject that contains information about the SSL session. This information can optionally be displayed to the user to provide details about the session.
timedout | A boolean value that indicates the reason for the callback. A value of true indicates that the PIN timed out and must be obtained again. A value of false indicates that the PIN was not specified at the time of the SSL connection.

Return value
String. Returns the PIN specified by the user.

Usage
A PowerBuilder application does not usually call the GetPin function directly. GetPin is called by EAServer when an EAServer client has not specified a PIN for logging in to a PKCS 11 token for an SSL connection.

To override the behavior of any of the functions of the SSLCallback object, create a standard class user object that descends from SSLCallback and customize this object as necessary. To let EAServer know which object to use when a callback is required, specify the name of the object in the callbackImpl SSL property. You can set this property value by calling the SetGlobalProperty function.

If you do not provide an implementation of GetPin, EAServer receives the CORBA::NO_IMPLEMENT exception and an empty string is returned. To obtain a useful return value, code the function to request the user to provide a PIN. You can supply information to the user such as the token name from the passed thesessioninfo object.

If an incorrect PIN or an empty string is returned, EAServer invokes the TrustVerify callback.

You can enable the user to cancel the attempt to connect by throwing an exception in this callback function. All exceptions thrown in SSLCallback functions return a CTSSecurity::UserAbortedException to the server. You need to catch the exception by wrapping the ConnectToServer function in a try-catch block.
GetPin

Examples

This example prompts the user to enter a PIN for a new SSL session or when a session has timed out. In practice you would want to replace the user’s entry in the text box with asterisks and allow the user more than one attempt to enter a correct PIN:

```powerbuilder
// instance variables
// string is_tokenName
// SSLServiceProvider issp_jag

CTSSecurity_sslSessionInfo  mySessionInfo
is_tokenName = mySessionInfo.getProperty( "tokenName" )
w_response w_pin

IF timedout THEN
    MessageBox("The SSL session has expired", &
            "Please reenter the PIN for access to the " + &
            ls_tokenName + " certificate database.")
ELSE
    MessageBox("The SSL session requires a PIN", &
            "Please enter the PIN for access to the " + &
            ls_tokenName + " certificate database.")
END IF

string s_PIN
userabortedexception ue_cancelled

// open prompt for PIN
Open(w_pin)
// get value entered
s_PIN = Message.StringParm

// set property if we're not to abort
if s_PIN <> ABORT_VALUE then
    issp_jag.setglobalproperty("pin", s_PIN)
else
    ue_cancelled = CREATE userabortedexception
    ue_cancelled.text = "User cancelled request when " &
    + "asked for PIN."
    throw ue_cancelled
end if
return s_PIN
```
GetRecordSet

Description
Returns the current ADO Recordset object.

Applies to
ADOResultSet objects

Syntax
`adoresultset.GetRecordSet ( adorecordsetobject )`

Argument | Description
---|---
`adoresultset` | An ADOResultSet object that contains an ADO Recordset.
`adorecordsetobject` | An OLEObject object into which the function places the current ADO Recordset. This argument is passed by reference.

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

Usage
Use the GetRecordSet function to return an ADO Recordset as an OLEObject object that can be used in PowerBuilder as a native ADO Recordset. The ADOResultSet object that contains the ADO Recordset must first have been populated using the SetRecordSet or SetResultSet function.

Examples
This example generates a result set in a ResultSet object from an existing DataStore object. The ResultSet object is used to populate a new ADOResultSet object. The GetRecordSet function on the ADOResultSet object is used to return an ADO Recordset as an OLEObject that can be used with ADO Recordset methods.

```
resultset lrs_resultset
ADOResultset lrs_ADOresultset
OLEObject loo_ADOrecordset
// Generate a result set from an existing DataStore
ds_source.GenerateResultSet(lrs_resultset)

// Create a new ADOResultSet object and populate it
// from the generated result set
lrs_ADOresultset = CREATE ADOResultSet
lrs_ADOresultset.SetResultSet(lrs_resultset)

// Pass the data in the ADOResultSet object
// to an OLEObject you can use as an ADO Recordset
```
GetRemote

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 have established.

**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.

Syntax

GetRemote ( location, target, applname, topicname [, bAnsi] )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>A string whose value is the location of the data you want returned from the DDE server application. The format of <strong>location</strong> depends on the particular DDE server application that will receive the message.</td>
</tr>
<tr>
<td>target</td>
<td>A string variable into which the returned data will be placed.</td>
</tr>
<tr>
<td>applname</td>
<td>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.</td>
</tr>
</tbody>
</table>
CHAPTER 10  PowerScript Functions

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 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.
GetRemote

Syntax

GetRemote ( location, target, handle [, windowhandle] [, bAnsi])

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.
bAnsi (optional) | A boolean identifying whether the string to get from the DDE server is in ANSI format. If bAnsi is NULL, false, or empty, PowerBuilder will first try to get the DDE data as a UNICODE formatted string. If bAnsi is true, PowerBuilder will try to get the DDE data as an ANSI formatted string.

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 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:

```powerscript
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

### GetSelectedDate

**Description**
Retrieves the selected date.

**Applies to**
MonthCalendar control

**Syntax**
```
controlname.GetSelectedDate(d)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the MonthCalendar control for which you want to get the selected date</td>
</tr>
<tr>
<td>d</td>
<td>A date value returned by reference that indicates the date selected</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 0 for success and one of the following negative values otherwise:
- `-1` A range of dates is selected
- `-2` Unknown failure

**Usage**
If a range of dates is selected, GetSelectedDate returns -1 and retrieves the earliest selected date.
**GetSelectedRange**

**Examples**

This example retrieves the selected date into `seldate`:

```powershell
integer li_return
Date seldate

li_return = mc_1.GetSelectedDate(seldate)
```

**See also**

GetSelectedRange
SetSelectedDate
SetSelectedRange

**GetSelectedRange**

**Description**

Retrieves the range of selected dates.

**Applies to**

MonthCalendar control

**Syntax**

`controlname.GetSelectedRange ( start, end )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the MonthCalendar control for which you want to determine the range of selected dates</td>
</tr>
<tr>
<td><code>start</code></td>
<td>A date value returned by reference that indicates the earliest date selected when a range of dates has been selected</td>
</tr>
<tr>
<td><code>end</code></td>
<td>A date value returned by reference that indicates the latest date selected when a range of dates has been selected</td>
</tr>
</tbody>
</table>

**Return value**

Integer. Returns 0 for success, -1 if only one date is selected, and -2 for an unknown failure.

**Usage**

If only one date is selected, `GetSelectedRange` returns -1 and the selected date is retrieved in the start and end parameters.

**Examples**

This code in the DateChanged event prompts the user to enter a second date after the first date in a range is entered, and then asks the user to confirm the range selected:

```powershell
date startdate, enddate
integer li_return

li_return = mc_1.GetSelectedRange(startdate, enddate)
if li_return = -1 then
    MessageBox("Selected Dates", &
        "Please select a return date")
elseif li_return = 0 then
```
MessageBox("Selected Dates", "You have selected "+ 
+ string(startdate) + " - " string(enddate) & 
+ "r\nClick OK to confirm")
else
    MessageBox("Selected Dates", & 
    "An error has occurred. Please reselect your " & 
+ "travel dates")
end if

See also
GetDisplayRange
GetSelectedDate
SetSelectedDate
SetSelectedRange

---

**GetSeriesLabelling**

**Description**
Determines whether the data for a given series is labeled in a DirectX 3D graph.

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

**Syntax**
```
controlname.GetSeriesLabelling ((graphcontrol,} series, value)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph from which you want data, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(DataWindow control only) [Optional] A string whose value is the name of the graph in the DataWindow control.</td>
</tr>
<tr>
<td>series</td>
<td>The string that names the series for which you want the series label setting.</td>
</tr>
<tr>
<td>value</td>
<td>A boolean passed by reference that indicates whether the series has a label.</td>
</tr>
</tbody>
</table>

**Return value**
Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, GetSeriesLabelling returns null.

**Usage**
GetSeriesLabelling retrieves the data from DirectX 3D Area, Bar, Col, or Line graphs. You cannot use this method with DirectX 3D Pie graphs.

**Examples**
These statements obtain the number of the series and datapoint for the graph gr_1 in the DataWindow control dw_employee and then set the series label.
integer SeriesNbr, ItemNbr
boolean refB
string ls_SeriesName
grObjectType clickedtype

// Get the number of the series and datapoint
clickedtype = this.ObjectAtPointer("gr_1", & SeriesNbr, ItemNbr)

// Get the name of series
ls_SeriesName = dw_employee.SeriesName("gr_1", & SeriesNbr)

// Set Series label
dw_employee.GetSeriesLabelling("gr_1", & ls_SeriesName, refB)

These statements obtain the number of the series and datapoint for the graph gr_1 and then set the series label.

integer SeriesNbr, ItemNbr
boolean refB
string ls_SeriesName
grObjectType clickedtype

clickedtype = gr_1.ObjectAtPointer(SeriesNbr, & ItemNbr)

ls_SeriesName = gr_1.SeriesName(SeriesNbr)

gr_1.GetSeriesLabelling(ls_SeriesName, refB)

See also GetDataLabelling
SetDataLabelling
SetSeriesLabelling

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.
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 do not 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
- **Syntax**: `controlname.GetSeriesStyle ( { graphcontrol, } seriesname, colortype, colorvariable )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the graph in which you want to obtain the color of a series, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td><code>graphcontrol</code></td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control for which you want the color of a series.</td>
</tr>
<tr>
<td><code>seriesname</code></td>
<td>A string whose value is the name of the series for which you want the color.</td>
</tr>
<tr>
<td><code>colortype</code></td>
<td>A string whose value is the name of the color you want to get.</td>
</tr>
<tr>
<td><code>colorvariable</code></td>
<td>A string whose value is the name of the variable to hold the color.</td>
</tr>
</tbody>
</table>

**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.
GetSeriesStyle

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colortype</td>
<td>A value of the grColorType enumerated datatype specifying the aspect of the series for which you want the color:</td>
</tr>
<tr>
<td></td>
<td>• Foreground! – Text color</td>
</tr>
<tr>
<td></td>
<td>• Background! – Background color</td>
</tr>
<tr>
<td></td>
<td>• LineColor! – Line color</td>
</tr>
<tr>
<td></td>
<td>• Shade! – Shade (for graphs that are 3-dimensional or have solid data markers)</td>
</tr>
<tr>
<td>colorvariable</td>
<td>A long variable in which you want to store the color's RGB value.</td>
</tr>
</tbody>
</table>

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:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
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)
```
CHAPTER 10  PowerScript Functions

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

Syntax
controlname.GetSeriesStyle ( { graphcontrol, } seriesname, linestyle, linewidth )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph for which you want the line style and width for a series in a graph, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control for which you want the line style information.</td>
</tr>
<tr>
<td>(DataWindow control only)</td>
<td></td>
</tr>
<tr>
<td>seriesname</td>
<td>A string whose value is the name of the series for which you want the line style information.</td>
</tr>
<tr>
<td>linestyle</td>
<td>A variable of type LineStyle in which you want to store the line style of seriesname.</td>
</tr>
<tr>
<td>linewidth</td>
<td>An integer variable in which you want to store the line width for seriesname. The width is measured in pixels.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. Stores in linestyle a value of the LineStyle enumerated datatype 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

PowerScript Reference 593
GetSeriesStyle

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

Syntax
controlname.GetSeriesStyle ( ( graphcontrol, ) seriesname, enumvariable )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph for which you want the style information for a series in a graph, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control for which you want the style information. (DataWindow control only)</td>
</tr>
<tr>
<td>seriesname</td>
<td>A string whose value is the name of the series for which you want the style information.</td>
</tr>
<tr>
<td>enumvariable</td>
<td>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.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. Stores in enumvariable a value of the appropriate enumerated datatype 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 datatype 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 datatype of *data_pattern* (the variable specified as the last argument) to find out the symbol type.
### 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

**Syntax**

```powerbuilder
controlname.GetSeriesStyle ( ( graphcontrol, ) seriesname, overlayindicator )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the graph for which you want the overlay status of a series in a graph, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td><code>graphcontrol</code></td>
<td>(DataWindow control only) A string whose value is the name of the graph in the DataWindow control for which you want the overlay status.</td>
</tr>
<tr>
<td><code>seriesname</code></td>
<td>A string whose value is the name of the series for which you want the overlay status.</td>
</tr>
<tr>
<td><code>overlayindicator</code></td>
<td>A boolean variable in which you want to store a value indicating whether the series is an overlay. GetSeriesStyle sets <code>overlayindicator</code> to <code>true</code> if the series is an overlay and <code>false</code> if it is not.</td>
</tr>
</tbody>
</table>

**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`:

```powerbuilder
boolean is_overlay
gr_emp_data.GetSeriesStyle(sle_series.Text, &
                          is_overlay)
```

---

GetSeriesStyle

**See also**
AddSeries
GetDataStyle
FindSeries
SetDataStyle
SetSeriesStyle
GetSeriesTransparency

Description
Obtains the transparency percentage of a series in a DirectX 3D graph (those with 3D rendering).

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

Syntax
controlname.GetSeriesTransparency( { graphcontrol, } series, transparency )

Argument | Description
--- | ---
controlname | The name of the graph from which you want series transparency data, or the name of 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 from which you want the series data.
series | The string that identifies the series from which you want the transparency value.
transparency | Integer value for percent transparency. A value of 0 means that the series is opaque and a value of 100 means that it is completely transparent.

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, GetSeriesTransparency returns null.

Usage
GetSeriesTransparency retrieves data from any DirectX 3D graph (those with 3D rendering).

Examples
These statements obtain the transparency value of the series named Costs in the graph gr_computers in the DataWindow control dw_equipment:

```powerscript
string SeriesName
integer rtn, ser_transp_value

// Get the number of the series.
SeriesNbr = dw_equipment.FindSeries( "gr_computers", "Costs")
rtn = dw_equipment.GetSeriesTransparency( "gr_computers", SeriesNbr, ser_transp_value)
```

These statements obtain the transparency value of the series named Income in the graph gr_1.

```powerscript
string SeriesName
integer rtn, ser_transp_value
```
GetShortName

Description
Gets the short name for the current PowerBuilder execution context.

Applies to
ContextInformation objects

Syntax
\[ servicereference.GetShortName( \text{shortname} ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicereference</td>
<td>Reference to the ContextInformation service instance.</td>
</tr>
<tr>
<td>shortname</td>
<td>String into which the function places the short name. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage
Call this function to determine the current execution environment.

The window plug-in and window ActiveX contexts are obsolete in the current version of PowerBuilder. For PowerBuilder 11.5 applications, the only value passed for the \text{shortname} argument is “PBRun”.

Examples
This example calls the GetShortName function. \text{ci} is an instance variable of type ContextInformation:

\[
\begin{align*}
\text{String } \text{ls\_name} \\
\text{this.GetContextService("ContextInformation", ci)} \\
\text{ci.GetShortName(ls\_name)} \\
\text{IF ls\_name <> "PBRun" THEN} \\
\text{cb\_close.visible = FALSE} \\
\text{END IF}
\end{align*}
\]

See also
GetCompanyName
GetContextService
GetFixesVersion
GetHostObject
GetSpacing

Description
Obtains the line spacing of the paragraph containing the insertion point in a
RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
rtename.GetSpacing()

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control in which you want to find out the line spacing of the paragraph containing the insertion point</td>
</tr>
</tbody>
</table>

Return value
Spacing. A value of the Spacing enumerated datatype 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 datatype spacing in the variable l_spacing. The value is the spacing for the paragraph with the insertion point:

```power_script
spacing l_spacing
l_spacing = rte_1.GetSpacing()
```

See also
GetTextStyle
SetSpacing
SetTextStyle

GetStatus

Description
Returns the status of the EAServer transaction associated with the calling thread.

Applies to
CORBACurrent objects
GetStatus

Syntax

`CORBACurrent.GetStatus()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CORBACurrent</code></td>
<td>Reference to the CORBACurrent service instance</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns -1 if an error occurs and one of the following positive integers if it succeeds:

1. Status active
2. Status marked rollback
3. Status prepared
4. Status committed
5. Status rolled back
6. Status unknown
7. Status no transaction
8. Status preparing
9. Status committing
10. Status rolling back

Usage

The GetStatus function can be used to determine the current status of a transaction by the client or component that initiated the transaction using the BeginTransaction function. EAServer must be using the two-phase commit transaction coordinator (OTS/XA).

GetStatus returns 1 when the transaction has started and no prepares have been issued.

When GetStatus returns 4 or 5, heuristics may exist; otherwise, the transaction would have been completed and destroyed and the value 7 returned.

A return value of 6 indicates that the transaction is in a transient condition and a subsequent call will eventually return another status. I

If GetStatus returns 8, 9, or 10, the transaction has begun but not yet completed the process of preparing, committing, or rolling back, probably because responses from participants in the transaction are pending.

Examples

This example shows the use of GetStatus to obtain the state of the current transaction:

```powershell
// Instance variable:
// CORBACurrent corbcurr
integer li_rc, li_status
```
li_rc = this.GetContextService("CORBACurrent", & corbcurr)
IF li_rc <> 1 THEN
   // handle the error
END IF
li_rc = corbcurr.Init("iiop://jagserver:2000")
IF li_rc <> 1 THEN
   // handle the error
ELSE
   li_status = corbcurr.GetStatus()
   CHOOSE CASE li_status
   CASE 1
      // take appropriate action for each value
      ...
   END CHOOSE
END IF

See also
BeginTransaction
CommitTransaction
GetContextService
GetTransactionName
Init
ResumeTransaction
RollbackOnly
RollbackTransaction
SetTimeout
SuspendTransaction

GetSyncRegistryProperties
Description Returns an integer to determine whether to use synchronization properties saved in the registry.
Applies to MLSyncrhonization, MLSync controls
Syntax syncObject.GetSyncRegistryProperties ()

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syncObject</td>
<td>The name of the synchronization object.</td>
</tr>
</tbody>
</table>
GetText

Return value

Integer. Returns 1 for success and -1 for failure. Failure occurs if SyncRegistryKey property is not set or if the key does not exist in the Windows registry.

Usage

The GetSyncRegistryProperties function sets synchronization object properties from values stored in the registry. The properties it can set include: AdditionalOpts, DownloadOnly, ExtendedOpts, Host, LogFileName, LogOpts, MLServerVersion, MLUser, ObjectRevision, Port, Publication, UploadOnly, UseLogFile, and UseWindow.

It cannot set secured properties such as AuthenticateParms, DBPass, and EncryptionKey that are never saved to the Windows registry.

Examples

The MLSync object generated by the MobiLink wizard for SQL Anywhere uses the following code in its Constructor event:

```powerbuilder
long rc
long RegistryRevision
RegistryRevision =this.GetObjectRevisionFromRegistry()
IF RegistryRevision < this.ObjectRevision THEN
  rc = this.SetSyncRegistryProperties()
ELSE
  rc = this.GetSyncRegistryProperties()
END IF
```

This code gets the values of authentication properties stored in the Windows registry only if the build number stored in the registry is higher than the build number of the running application.

See also

GetCommandString
GetObjectRevisionFromRegistry
SetParm
SetSyncRegistryProperties

GetText

Description

Returns the Value property as a text string with the specified Format or CustomFormat applied.

Applies to

DatePicker controls

Syntax

```
controlname.GetText ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the control for which you want to get the text</td>
</tr>
</tbody>
</table>
Return value String.

Usage Use the GetText function to return the date and time stored in the Value property as a text string formatted according to the Format property, or if Format is set to dtfCustom!, according to the format specified in the CustomFormat property.

Examples This example retrieves the date and time stored in the Value property of dp_1 to the string ls_text:

```powerscript
string ls_text
ls_text = dp_1.GetText()
```

See also GetValue
SetValue

GetTextColor

Description Obtains the color of selected text in a RichTextEdit control.

Applies to RichTextEdit controls

Syntax `rtename.GetTextColor()`

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:

```powerscript
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
\texttt{rtename.GetTextStyle (textstyle)}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{rtename}</td>
<td>The name of the RichTextEdit control in which you want to find the formatting of selected text.</td>
</tr>
<tr>
<td>\texttt{textstyle}</td>
<td>A value of the enumerated datatype TextStyle specifying the text style you want to check for. Values are: Bold!Italic!Strikeout!Subscript!Superscript!Underlined!</td>
</tr>
</tbody>
</table>

Return value
Boolean. Returns true if the selected text is formatted with the specified text style and false if it is not. If \texttt{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 \texttt{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:

\begin{verbatim}
istr_text.b_isBold = rte_fancy.GetTextStyle(Bold!)
istr_text.b_isItalic = rte_fancy.GetTextStyle(Italic!)
istr_text.b_isUnderlined = &
    rte_fancy.GetTextStyle(Underlined!)
\end{verbatim}

See also
GetTextColor
SetSpacing
SetTextColor
SetTextStyle
CHAPTER 10  PowerScript Functions

GetToday

Description  Returns the value that the calendar uses as today’s date.
Applies to  DatePicker, MonthCalendar controls
Syntax  \textit{controlname}\textunderscore\textit{GetToday}( )

\begin{center}
\begin{tabular}{l|l}
\textbf{Argument} & \textbf{Description} \\
\hline
\textit{controlname} & The name of the control for which you want to get today’s date \\
\end{tabular}
\end{center}

Return value  Date.
Usage  By default, the current system date is set as the Today date. You can use the \textit{SetToday} function to specify a different date.
Examples  This example retrieves the Today date in a DatePicker control into \textit{currentdate}:
\begin{verbatim}
Date currentdate
currentdate = dp_1.GetToday()
\end{verbatim}

See also  SetToday

GetToolbar

Description  Gets the current values for alignment, visibility, and title of the specified toolbar.
Applies to  MDI frame and sheet windows
Syntax  \textit{window}\textunderscore\textit{GetToolbar}( \textit{toolbarindex}, \textit{visible} \{, \textit{alignment} \{, \textit{floatingtitle} \} \} )

\begin{center}
\begin{tabular}{l|l}
\textbf{Argument} & \textbf{Description} \\
\hline
\textit{window} & The MDI frame or sheet to which the toolbar belongs \\
\textit{toolbarindex} & An integer whose value is the index of the toolbar for which you want the current settings \\
\textit{visible} & A boolean variable in which you want to store a value indicating whether the toolbar is visible \\
\textit{alignment} & A variable of the ToolbarAlignment enumerated datatype in which you want to store the current alignment of the toolbar \textbf{(optional)} \\
\textit{floatingtitle} & A string variable in which you want to store the toolbar title that is displayed when the alignment is Floating! \textbf{(optional)} \\
\end{tabular}
\end{center}
GetToolbar

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 \( lb_{\text{visible}} \), \( lta_{\text{align}} \), and \( ls_{\text{title}} \):

```powershell
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 \( sle_{\text{index}} \). The IF and CHOOSE CASE statements convert the values to strings so they can be displayed in mle_toolbar:

```powershell
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" 
  + la_title

See also
GetToolbarPos
SetToolbar
SetToolbarPos

GetToolbarPos

Gets position information for the specified toolbar.

<table>
<thead>
<tr>
<th>To get</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docking position of a docked toolbar</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Coordinates and size of a floating toolbar</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

For docked toolbars

Gets the position of a docked toolbar.

Description

Applies to MDI frame and sheet windows

Syntax `window.GetToolbarPos ( toolbarindex, dockrow, offset )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>window</td>
<td>The MDI frame or sheet to which the toolbar belongs.</td>
</tr>
<tr>
<td>toolbarindex</td>
<td>An integer whose value is the index of the toolbar for which you want the current settings.</td>
</tr>
<tr>
<td>dockrow</td>
<td>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.</td>
</tr>
<tr>
<td>offset</td>
<td>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.</td>
</tr>
</tbody>
</table>
GetToolbarPos

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, GetToolbarPos 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 $le_2. The example gets the toolbar position information and displays it in a MultiLineEdit $mle_1:

```powerbuilder
integer li_index, li_rtn
integer li_dockrow, li_offset

li_index = Integer($le_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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>window</code></td>
<td>The MDI frame or sheet to which the toolbar belongs.</td>
</tr>
<tr>
<td><code>toolbarindex</code></td>
<td>An integer whose value is the index of the toolbar for which you want the current settings.</td>
</tr>
<tr>
<td><code>x</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>y</code></td>
<td></td>
</tr>
<tr>
<td><code>width</code></td>
<td></td>
</tr>
<tr>
<td><code>height</code></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 10  PowerScript Functions

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:

```powerShell
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
GetTransactionName

Description
Returns a string describing the EAServer transaction associated with the calling thread.

Applies to
CORBACurrent objects

Syntax
CORBACurrent.GetTransactionName()

Return value
String. Returns a printable string describing the transaction if a transaction exists and an empty string otherwise.

Usage
The GetTransactionName function returns a string identifying the transaction associated with the calling thread. This string is typically used for debugging.

GetTransactionName can be called by a client or a component that is marked as OTS style. EAServer must be using the two-phase commit transaction coordinator (OTS/XA).

Examples
This example shows the use of GetTransactionName to return information about a transaction to a client:

    // Instance variables:
    // CORBACurrent corbcurr
    string ls_transacname

    // Get an instance of the CORBACurrent object
    // and initialize it
    ...
    ls_transacname = corbcurr.GetTransactionName()
    MessageBox("Transaction Name", ls_transacname)

See also
BeginTransaction
CommitTransaction
GetContextService
GetStatus
Init
ResumeTransaction
RollbackOnly
RollbackTransaction
SetTimeout
SuspendTransaction
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 more information on the InternetResult standard class user object and the InternetData function, use the PowerBuilder Browser.

Examples
This example calls the GetURL function./inet_base is an instance variable of type inet:

  iir_msgbox = CREATE n_ir_msgbox
  inet_base.GetURL( sle_url.text, iir_msgbox )

See also
HyperLinkToURL
InternetData
PostURL
**GetValue**

Returns the date and time in the Value property of the control.

**Apply to**
DatePicker control

**Syntax**
```
controlname.GetValue (d, t)
controlname.GetValue (dt)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the control for which you want to get the date and time</td>
</tr>
<tr>
<td>d</td>
<td>The date value in the Value property returned by reference</td>
</tr>
<tr>
<td>t</td>
<td>The time value in the Value property returned by reference</td>
</tr>
<tr>
<td>dt</td>
<td>The DateTime value in the Value property returned by reference</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 1 for success and one of the following negative values for failure:

- -1 Invalid date and/or time values
- -2 Other error

**Usage**
The GetValue function can return the date and time parts of the Value property in separate date and time variables or a single DateTime variable.

**Examples**
In this example, the GetValue function is called twice, once to return separate date and time values and once to return a DateTime value. The values returned are written to a multiline edit control:
```
date d
time t
datetime dt
integer li_ret1, li_ret2

li_ret1 = dp_1.GetValue(d, t)
li_ret2 = dp_1.GetValue(dt)

mle_1.text += string(d) + " -r-n"
mle_1.text += string(t) + " -r-n"
mle_1.text += string(dt) + " -r-n"
```

**See also**
GetText
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 8.0.3).

Applies to
ContextInformation objects

Syntax
`servicereference.GetVersionName( name )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicereference</td>
<td>Reference to the ContextInformation service instance.</td>
</tr>
<tr>
<td>name</td>
<td>String into which the function places the version name. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

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. `ci` is an instance variable of type ContextInformation:

```powerscript
String ls_name
String ls_version
Constant String ls_currver = "8.0.3"

GetContextService("ContextInformation", ci)
ci.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
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.

Syntax
Handle ( objectname {, previous } )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectname</td>
<td>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.</td>
</tr>
<tr>
<td>previous (optional)</td>
<td>(Obsolete argument) A boolean indicating whether you want the handle of the previous instance of an application. You can use this argument with the Application object only. In current versions of Windows, Handle always returns 0 when this argument is set to true.</td>
</tr>
</tbody>
</table>

Return value
Long. Returns the handle of objectname. If objectname is an application and previous is true, Handle always returns 0.

If objectname cannot be referenced at runtime, 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.

If you ask for the handle of a previous instance of an application by setting the previous flag to true, Handle always returns 0 in current versions of Windows. Use the Windows FindWindow function to determine whether an instance of the application’s main window is already open.

Examples
This statement returns the handle to the window w_child:

Handle (w_child)
These statements use an external function called FlashWindow to change the title bar of a window to inactive and then return it to active. The external function declaration is:

```plaintext
function boolean flashwindow(uint hnd, boolean inst) &
   library "user.exe"
```

The code that flashes the window’s title bar is:

```plaintext
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)
```

For applications, the Handle function does not return a useful value when the previous flag is true. You can use the FindWindow Windows function to determine whether a Windows application is already running. FindWindow returns the handle of a window with a given title.

Declare FindWindow and SetForegroundWindow as global external functions:

```plaintext
PUBLIC FUNCTION unsignedlong FindWindow (long &
   classname, string windowname) LIBRARY "user32.dll" &
   ALIAS FOR FindWindowW
PUBLIC FUNCTION int SetForegroundWindow (unsignedlong &
   hwnd) LIBRARY "user32.dll" ALIAS FOR &
   SetForegroundWindowW
```

Then add code like the following to your application’s Open event:

```plaintext
unsignedlong hwnd

hwnd = FindWindow( 0, "Main Window")
if hwnd = 0 then
   // no previous instance, so open the main window
   open( w_main )
else
   // open the previous instance window and halt
   SetForegroundWindow( hwnd )
   HALT CLOSE
end if
```

See also

Send
**Hide**

**Description**
Makes an object or control invisible. Users cannot interact with an invisible object. It does not respond to any events, so the object is also, in effect, disabled.

**Applies to**
Any object

**Syntax**
```
objectname.Hide()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectname</td>
<td>The name of the object or control you want to make invisible</td>
</tr>
</tbody>
</table>

**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 drop-down 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 does not respond to events, but is still visible, set its `Enabled` property.

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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>The time from which you want to obtain the hour</td>
</tr>
</tbody>
</table>

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 method for DataWindows in the DataWindow Reference or online Help

HyperLinkToURL

Description
Opens the default Web browser, displaying the specified URL.

Applies to
Inet objects

Syntax
servicereference. HyperlinkToURL ( url )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicereference</td>
<td>Reference to the Internet service instance</td>
</tr>
<tr>
<td>url</td>
<td>String specifying the URL to open in the default Web browser</td>
</tr>
</tbody>
</table>

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. inet_base is an instance variable of type inet:

GetContextService( "Internet" , inet_base )

inet_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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>The number of seconds of user inactivity allowed before PowerBuilder triggers an Application Idle event. A value of 0 terminates Idle detection.</td>
</tr>
</tbody>
</table>

**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:

\texttt{Idle(300)}

This statement turns off idle detection:

\texttt{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:

\texttt{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:

\begin{verbatim}
// Statements to set the database to the desired state
...
Restart() // Restarts the application
\end{verbatim}

See also

Restart
Timer

ImpersonateClient

Description

Allows a COM object running on COM+ to take on the security attributes of the client for the duration of a call.

Applies to

TransactionServer objects

Syntax

\texttt{transactionserver.ImpersonateClient ()}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{transactionserver}</td>
<td>Reference to the TransactionServer service instance</td>
</tr>
</tbody>
</table>

Return value

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

Usage

ImpersonateClient allows a COM object to run in the client’s security context for the duration of a call. Running in the client’s security context gives the server process access to the same resources as the client. This can either restrict or expand the server’s access to resources. For example, if the client does not have update rights to a database but the server does, impersonating the client before accessing the database prevents the client from updating the database.
After completing the processing that requires the client’s security context, call
RevertToSelf to revert to the server’s security context.

**Examples**

This example creates an instance of the transaction server context object and
impersonates the client to perform some processing:

```powerbuilder
TransactionServer txninfo_test
integer li_rc
li_rc = GetContextService( "TransactionServer", &
    txninfo_test )
// Handle error if necessary

// Impersonate the client
txninfo_test.ImpersonateClient()
// Perform processing with client security context
...
// Revert to server’s security context
txninfo_test.RevertToSelf()
```

**See also**
IsCallerInRole
IsImpersonating
IsSecurityEnabled
RevertToSelf

---

**ImportClipboard**

**Description**

Inserts data into a DataWindow control, DataStore object, or graph control
from tab-separated, comma-separated, or XML data on the clipboard.

For DataWindow and DataStore syntax, see the ImportClipboard method for
DataWindows in the *DataWindow Reference* or the online Help.

**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**

```powerbuilder
graphname.ImportClipboard( { importtype }, { startrow {, endrow {, startcolumn } } } )
```
### Return value

Returns the number of rows that were imported if it succeeds and one of the following negative integers if an error occurs:

- **-1** No rows or `startrow` value supplied is greater than the number of rows in the string
- **-2** Input data does not match number of columns or required column type
- **-3** Invalid argument
- **-4** Invalid input

### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>importtype</code> (optional)</td>
<td>An enumerated value of the SaveAsType DataWindow constant. Valid type arguments for <code>ImportClipboard</code> are: Text! CSV! XML! If you want to generate an XML trace file, the XML! argument is required.</td>
</tr>
<tr>
<td><code>graphname</code></td>
<td>The name of the graph control to which you want to copy data from the clipboard.</td>
</tr>
<tr>
<td><code>startrow</code> (optional)</td>
<td>The number of the first detail row in the clipboard that you want to copy. The default is 1. For default XML import, if <code>startrow</code> is supplied, the first ( N ) ((startrow - 1)) elements are skipped, where ( N ) is the DataWindow row size. For template XML import, if <code>startrow</code> is supplied, the first ((startrow - 1)) occurrences of the repetitive row mapping defined in the template are skipped.</td>
</tr>
<tr>
<td><code>endrow</code> (optional)</td>
<td>The number of the last detail row in the clipboard that you want to copy. The default is the rest of the rows. For default XML import, if <code>endrow</code> is supplied, import stops when ( N \times endrow ) elements have been imported, where ( N ) is the DataWindow row size. For template XML import, if <code>endrow</code> is supplied, import stops after <code>endrow</code> occurrences of the repetitive row mapping defined in the template have been imported.</td>
</tr>
<tr>
<td><code>startcolumn</code> (optional)</td>
<td>The number of the first column in the clipboard that you want to copy. The default is 1. For default XML import, if <code>startcolumn</code> is supplied, import skips the first ((startcolumn - 1)) elements in each row. This argument has no effect on template XML import.</td>
</tr>
</tbody>
</table>
-11 XML Parsing Error; XML parser libraries not found, or XML not well formed

-12 XML Template does not exist or does not match the DataWindow

If any argument’s value is null, ImportClipboard returns null. If the optional importtype argument is specified and is not a valid type, ImportClipboard returns -3.

Usage

The clipboard data must be formatted in tab-separated or comma-separated columns or in XML. The datatypes and order of the DataWindow object’s columns must match the data on the clipboard.

For graphs, ImportClipboard uses only 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 does not 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:

\[\text{gr\_employee}.\text{ImportClipboard}(2, 30, 3)\]

See also
ImportFile
ImportString

### ImportFile

**Description**

Inserts data into a DataWindow control, DataStore object, or graph control from data in a file. The data can be tab-separated text, comma-separated text, XML, 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.

For DataWindow and DataStore syntax, see the `ImportFile` method for DataWindows in the *DataWindow Reference* or the online Help.

**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**

\[\text{graphname}.\text{ImportFile} (\{ \text{importtype} \}, \text{filename} \{, \text{startrow} \{, \text{endrow} \{, \text{startcolumn} \} \}\} )\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>graphname</code></td>
<td>The name of the graph control to which you want to copy data from the specified file.</td>
</tr>
<tr>
<td><code>importtype</code> (optional)</td>
<td>An enumerated value of the SaveAsType DataWindow constant. If this argument is specified, the <code>importtype</code> argument can be specified without an extension. Valid type arguments for <code>ImportFile</code> are:</td>
</tr>
<tr>
<td></td>
<td>Text!</td>
</tr>
<tr>
<td></td>
<td>CSV!</td>
</tr>
<tr>
<td></td>
<td>XML!</td>
</tr>
<tr>
<td></td>
<td>DBase2!</td>
</tr>
<tr>
<td></td>
<td>DBase3!</td>
</tr>
<tr>
<td><code>filename</code></td>
<td>A string whose value is the name of the file from which you want to copy data. The file must be an ASCII, tab-separated file (TXT), comma-separated file (CSV), Extensible , or dBase format 2 or 3 file (DBF). Specify the file's full name. If the optional <code>importtype</code> is not specified, the name must end in the appropriate extension. If you do not specify <code>filename</code> or if it is null, <code>ImportFile</code> prompts the user for a file name. The remaining arguments are ignored.</td>
</tr>
</tbody>
</table>
**ImportFile**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>startrow</strong> (optional)</td>
<td>The number of the first detail row in the file that you want to copy. The default is 1. For default XML import, if <strong>startrow</strong> is supplied, the first (N) ((\text{startrow} - 1)) elements are skipped, where (N) is the DataWindow row size. For template XML import, if <strong>startrow</strong> is supplied, the first ((\text{startrow} - 1)) occurrences of the repetitive row mapping defined in the template are skipped.</td>
</tr>
<tr>
<td><strong>endrow</strong> (optional)</td>
<td>The number of the last detail row in the file that you want to copy. The default is the rest of the rows. For default XML import, if <strong>endrow</strong> is supplied, import stops when (N \times \text{endrow}) elements have been imported, where (N) is the DataWindow row size. For template XML import, if <strong>endrow</strong> is supplied, import stops after <strong>endrow</strong> occurrences of the repetitive row mapping defined in the template have been imported.</td>
</tr>
<tr>
<td><strong>startcolumn</strong> (optional)</td>
<td>The number of the first column in the file that you want to copy. The default is 1. For default XML import, if <strong>startcolumn</strong> is supplied, import skips the first ((\text{startcolumn} - 1)) elements in each row. This argument has no effect on template XML import.</td>
</tr>
</tbody>
</table>

**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** No rows or **startrow** value supplied is greater than the number of rows in the file
- **2** Empty file or input data does not match number of columns or required column type
- **3** Invalid argument
- **4** Invalid input
- **5** Could not open the file
- **6** Could not close the file
- **7** Error reading the text
- **8** Unsupported file name suffix (must be *.txt, *.csv, *.dbf or *.xml)
- **10** Unsupported dBase file format (not version 2 or 3)
CHAPTER 10  PowerScript Functions

-11 XML Parsing Error; XML parser libraries not found or XML not well formed

-12 XML Template does not exist or does not match the DataWindow

If any argument’s value is null, ImportFile returns null. If the optional importtype argument is specified and is not a valid type, ImportFile returns -3.

The format of the file can be indicated by specifying the optional importtype parameter, or by including the appropriate file extension.

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 box.

Double quotes The location and number of double quote marks in a field in a tab delimited file affect how they are handled when the file is imported. If a string is enclosed in one pair of double quotes, the quotes are discarded. If it is enclosed in three pairs of double quotes, one pair is retained when the string is imported. If the string is enclosed in two pairs of double quotes, the first pair is considered to enclose a null string, and the rest of the string is discarded.

When there is a double quote at the beginning of a string, any characters after the second double quote are discarded. If there is no second double quote, the tab character delimiting the fields is not recognized as a field separator and all characters up to the next occurrence of a double quote, including a carriage return, are considered to be part of the string. A validation error is generated if the combined strings exceed the length of the first string.
Double quotes after the first character in the string are rendered literally. Here are some examples of how tab-delimited strings are imported into a two-column DataWindow:

<table>
<thead>
<tr>
<th>Text in file</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Joe&quot; TAB &quot;Donaldson&quot;</td>
<td>Joe Donaldson</td>
</tr>
<tr>
<td>Bernice TAB &quot;&quot;Ramakrishnan&quot;</td>
<td>Bernice &quot;Ramakrishnan&quot;</td>
</tr>
<tr>
<td>&quot;&quot;Mary&quot;&quot; TAB &quot;&quot;Li&quot;&quot;</td>
<td>Empty cells</td>
</tr>
<tr>
<td>&quot;Mich&quot;ael TAB &quot;&quot;Lopes&quot;&quot;</td>
<td>Mich &quot;Lopes&quot;</td>
</tr>
<tr>
<td>&quot;Amy TAB Doherty&quot;</td>
<td>Amy&lt;TAB&gt;Doherty in first cell, second cell empty</td>
</tr>
<tr>
<td>3&quot;&quot;&quot;&quot; TAB 4&quot;&quot;</td>
<td>3&quot;&quot;&quot;&quot; 4&quot;&quot;</td>
</tr>
</tbody>
</table>

**Specifying a null string for file name**
If you specify a null string for `filename`, the remaining arguments are ignored. All the rows and columns in the file are imported.

**Examples**
This statement copies all the data in the file `D:\EMPLOYEE.TXT` to `gr_employee` starting at the first row:

```powerbuilder
gr_employee.ImportFile("D:\EMPLOYEE.TXT")
```

This statement copies the data from the file `D:\EMPLOYEE.TXT` starting with row 2 column 3 and ending with row 30 column 5 to the graph `gr_employee`:

```powerbuilder
gr_employee.ImportFile("D:\EMPLOYEE.TXT", 2, 30, 3)
```

The following statements are equivalent. Both import the contents of the XML file named `myxmldata`:

```powerbuilder
gr_control.ImportFile(myxmldata.xml)
gcontrol.ImportFile(XML!, myxmldata)
```

This example causes PowerBuilder to display the Specify Import File dialog box:

```powerbuilder
string null_str
SetNull(null_str)
dw_main.ImportFile(null_str)
```

**See also**
ImportClipboard
ImportString
**ImportString**

**Description**

Inserts data into a DataWindow control, DataStore object, or graph control from tab-separated, comma-separated, or XML 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.

For DataWindow and DataStore syntax, see the `ImportString` method for DataWindows in the *DataWindow Reference* or the online Help.

**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 ( { importtype }, string {, startrow {, endrow {, startcolumn } } } )
```

**Argument** | **Description**
--- | ---
`graphname` | The name of the graph control to which you want to copy data from the specified string.

`importtype` (optional) | A value of the `SaveAsType` enumerated datatype (PowerBuilder) or a string (Web DataWindow) specifying the format of the imported string. Valid type arguments are:

- Text!
- CSV!
- XML!

If you want to generate an XML trace file, the XML! argument is required.

`string` | A string from which you want to copy the data. The string should contain tab-separated or comma-separated columns or XML with one row per line (see Usage).

`startrow` (optional) | The number of the first detail row in the string that you want to copy. The default is 1.

For default XML import, if `startrow` is supplied, the first \( N \) (
\( startrow \cdot 1 \)) elements are skipped, where \( N \) is the DataWindow row size.

For template XML import, if `startrow` is supplied, the first \( (startrow \cdot 1) \) occurrences of the repetitive row mapping defined in the template are skipped.
**ImportString**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>endrow</code></td>
<td>The number of the last detail row in the string that you want to copy. The default is the rest of the rows. For default XML import, if <code>endrow</code> is supplied, import stops when ( N \times \text{endrow} ) elements have been imported, where ( N ) is the DataWindow row size. For template XML import, if <code>endrow</code> is supplied, import stops after <code>endrow</code> occurrences of the repetitive row mapping defined in the template have been imported.</td>
</tr>
<tr>
<td><code>startcolumn</code></td>
<td>The number of the first column in the string that you want to copy. The default is 1. For default XML import, if <code>startcolumn</code> is supplied, import skips the first ( \text{startcolumn} - 1 ) elements in each row. This argument has no effect on template XML import.</td>
</tr>
</tbody>
</table>

**Return value**

Returns the number of data points that were imported if it succeeds and one of the following negative integers if an error occurs:

-1 No rows or startrow value supplied is greater than the number of rows in the string

-2 Empty string or input data does not match number of columns or required column type

-3 Invalid argument

-4 Invalid input

-11 XML Parsing Error; XML parser libraries not found or XML not well formed

-12 XML Template does not exist or does not match the DataWindow

If any argument’s value is null, `ImportString` returns null. If the optional `importtype` argument is specified and is not a valid type, `ImportString` returns -3.

**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.

---

628 PowerBuilder
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 \texttt{ls\_Text} starting with row 2 column 3 and ending with row 30 column 5 to the graph \texttt{gr\_employee}:

```
string ls\_Text
ls\_Text = . . .
gr\_employee.\textbf{ImportString}(ls\_Text, 2, 30, 3)
```

The following script stores data for two series in the string \texttt{ls\_gr} and imports the data into the graph \texttt{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.\textbf{ImportString}(ls\_gr, 1)
```

See also

ImportClipboard
ImportFile

### \textbf{IncomingCallList}

**Description**

Provides a list of the callers of a routine included in a performance analysis model.

**Applies to**

ProfileRoutine object

**Syntax**

\begin{verbatim}
\textit{instancetype}.\textbf{IncomingCallList} \textit{( list, aggregateduplicateroutinecalls )}
\end{verbatim}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{instancetype}</td>
<td>Instance name of the ProfileRoutine object.</td>
</tr>
<tr>
<td>\textit{list}</td>
<td>An unbounded array variable of datatype ProfileCall in which IncomingCallList stores a ProfileCall object for each caller of the routine. This argument is passed by reference.</td>
</tr>
</tbody>
</table>
IncomingCallList

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregated duplicateroutinecalls</td>
<td>A boolean indicating whether duplicate routine calls will result in the creation of a single or of multiple ProfileCall objects.</td>
</tr>
</tbody>
</table>

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 aggregated duplicateroutinecalls 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 aggregated duplicateroutinecalls is true, a new ProfileCall object is created that aggregates all calls from the calling routine to the current routine. If aggregated duplicateroutinecalls 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 that called each routine:

```powerbuilder
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
Init

Sets ORB property values or initializes an instance of the CORBACurrent service object.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set ORB property values for client connections to EAServer using the JaguarORB object</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Initialize an instance of the CORBACurrent service object for client- or component-managed transactions</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

**For setting ORB property values**

Description Sets ORB property values. This function is used by PowerBuilder clients connecting to EAServer.

Applies to JaguarORB objects

Syntax

```
jaguarorb.Init ( options )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jaguarorb</code></td>
<td>An instance of JaguarORB.</td>
</tr>
<tr>
<td><code>options</code></td>
<td>A string that specifies one or more ORB property values. If you specify multiple property values, you need to separate the property values with commas. For a complete list of supported ORB properties, see the online Help for the Options property of the Connection object.</td>
</tr>
</tbody>
</table>

Return value Long. Returns 0 if it succeeds and a negative number if an error occurs.

Usage ORB properties configure settings required by the EAServer ORB driver.

You do not need to call the Init function to use the JaguarORB object. If you do not call Init, the EAServer ORB driver uses the default property values.

The Init function can be called multiple times on the same JaguarORB object. PowerBuilder creates a new internal instance of the JaguarORB object the first time and uses this object for all subsequent calls.

For additional examples, see the functions on the See also list.

Examples The following example shows the use of the Init function to set the RetryCount and RetryDelay ORB properties:

```
JaguarORB my_orb
CORBAObject my_corbaobj
...```


my_orb = CREATE JaguarORB
my_orb.Init("ORBRetryCount=3, ORBRetryDelay=1000")

See also
Object_To_String
Resolve_Initial_References
String_To_Object

Syntax 2

For initializing CORBACurrent

Description
Initializes an instance of the CORBACurrent service object.

Applies to
CORBACurrent objects

Syntax
`CORBACurrent.Init ( { connection | URL } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CORBACurrent</code></td>
<td>Reference to the CORBACurrent service instance.</td>
</tr>
<tr>
<td><code>connection</code></td>
<td>The name of the Connection object for which a connection has already been established to a valid EAServer host. Either <code>connection</code> or <code>URL</code> is required if the Init function is called by a client.</td>
</tr>
<tr>
<td><code>URL</code></td>
<td>String. The name of a URL that identifies a valid EAServer host. Either <code>connection</code> or <code>URL</code> is required if the Init function is called by a client.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 0 if it succeeds and one of the following values if the service object could not be initialized:

-1  Unknown error

-2  Service object not running in EAServer (no argument) or Connection object not connected to EAServer (argument is Connection object)

-3  ORB initialization error

-4  Error on a call to the `ORB.resolve_initial_references("TransactionCurrent")` method

-5  Error on a call to the narrow method
Usage

The Init function can be called from a PowerBuilder component running in EAServer whose transaction property is marked as OTS style, or by a PowerBuilder client. The Init function must be called to initialize the CORBACurrent object before any other functions are called. EAServer must be using the two-phase commit transaction coordinator (OTS/XA) and a reference to the CORBACurrent object must first be obtained using the GetContextService function.

When Init is called from a PowerBuilder component running in EAServer, no arguments are required. If the calling component is not marked as OTS style, the CORBACurrent object is not initialized.

When Init is called from a PowerBuilder client and the client is responsible for the transaction, the CORBACurrent object must be initialized by calling Init with either a Connection object or a URL string as the argument. In the case of a Connection object, the client must already be connected to a valid EAServer host using that Connection object. Using a Connection object is preferred because the code is more portable.

Examples

This example shows the use of Init in a PowerBuilder EAServer component to initialize an instance of the CORBACurrent object:

```powerbuilder
// Instance variables:
// CORBACurrent corbcurr
int li_rc

li_rc = this.GetContextService("CORBACurrent",
corbcurr)
IF li_rc <> 1 THEN
   // handle the error
ELSE
   li_rc = corbcurr.init()
   IF li_rc <> 0 THEN
      // handle the error
   END IF
END IF
```

In this example, Init is called by a PowerBuilder client application that has already connected to EAServer using the myconn Connection object and has created a reference called corbcurr to the CORBACurrent object:

```powerbuilder
li_rc = corbcurr.init( myconn )
IF li_rc <> 0 THEN
   // handle the error
END IF
```
In this example, the PowerBuilder client application calls the `init` function using a valid URL:

```powershell
li_rc = corbcurr.init( "iiop://localhost:2000" )
IF li_rc <> 0 THEN
   // handle the error
END IF
```

InputFieldChangeData

Description

Modifies the data value of input fields in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

```
rtename.InputFieldChangeData( inputfieldname, inputfieldvalue )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</code></td>
<td>The name of the RichTextEdit control in which you want to change the data in the specified input fields.</td>
</tr>
<tr>
<td><code>inputfieldname</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>inputfieldvalue</code></td>
<td>A string whose value is the data to be assigned to the specified input fields.</td>
</tr>
</tbody>
</table>

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.

See also

BeginTransaction
CommitTransaction
GetContextService
getStatus
GetTransactionName
ResumeTransaction
RollbackOnly
RollbackTransaction
SetTimeout
SuspendTransaction
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`:

```power_script
integer rtn
rtn = rte_1.InputFieldChangeData &
       ("instrument", lb_instruments.SelectedItem())

st_status.Text = String(rtn)
```

If the text in `rte_1` looks like this:

```plaintext
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:

```plaintext
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 ( )
```
**InputFieldDeleteCurrent**

**Description**
Deletes the input field that is selected in a RichTextEdit control.

**Applies to**
RichTextEdit controls

**Syntax**
```powershell
rtename.InputFieldDeleteCurrent() - 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.

---

**InputFieldCurrentName**

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control in which you want to get the input field’s name</td>
</tr>
</tbody>
</table>

**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:

```powershell
g string ls_inputname
   ls_inputname = rte_1.InputFieldCurrentName()
```

**See also**
InputFieldChangeData
InputFieldDeleteCurrent
InputFieldGetData
InputFieldInsert
InputFieldLocate
DataSource
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:

```powerscript
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control in which you want to get data from the selected input field</td>
</tr>
<tr>
<td>inputfieldname</td>
<td>A string whose value is the name of input field from which you want to get the data</td>
</tr>
</tbody>
</table>

Return value
String. The data in the input field. InputFieldGetData returns the empty string ("") if the field does not exist or an error occurs.

Examples
This example gets the data in the input field empname:

```powerscript
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</code></td>
<td>The name of the RichTextEdit control in which you want to insert an input field</td>
</tr>
<tr>
<td><code>inputfieldname</code></td>
<td>A string whose value is the name of input field to be inserted. The name does not have to be unique</td>
</tr>
</tbody>
</table>

**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.

**Examples**
If there is a selection, `InputFieldInsert` inserts the field at the beginning of the selection. The input field and the selection remain selected:
```
  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 } )
```

---

638  
PowerBuilder
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:

```power
string ls_name
ls_name = rte_1.InputFieldLocate(Next!)
```

This example locates the last input field in the document:

```power
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":

```power
string ls_name
ls_name = rte_1.InputFieldLocate(Last!, "address")
```
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph into which you want to insert a category.</td>
</tr>
<tr>
<td>categoryvalue</td>
<td>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 datatype as the datatype of the category axis.</td>
</tr>
<tr>
<td>categorynumber</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 datatype 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.
CHAPTER 10 PowerScript Functions

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:

```plaintext
gr_data.InsertCategory("Qty", 0)
```

is equivalent to:

```plaintext
gr_data.AddCategory("Qty")
```

**Examples**

These statements insert a category called Macs before the category named PCs in the graph `gr_product_data`:

```plaintext
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 does not matter:

```plaintext
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.

Syntax
```
ole2control.InsertClass ( classname )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ole2control</td>
<td>The name of the OLE control in which you want to create a new object</td>
</tr>
<tr>
<td>classname</td>
<td>A string whose value is the name of the class of the object you want to create</td>
</tr>
</tbody>
</table>

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 )
```
InsertColumn

Description

Inserts a column anywhere in the control. If the index you specify is greater than the current number of columns, the column is inserted after the last column.

Usage

You can insert a column anywhere in the control. If the index you specify is greater than the current number of columns, the column is inserted after the last 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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>seriesnumber</code></td>
<td>The number that identifies the series in which you want to insert data.</td>
</tr>
<tr>
<td><code>datapoint</code></td>
<td>The number of the data point before which you want to insert the data.</td>
</tr>
<tr>
<td><code>datavalue</code></td>
<td>The value of the data point you want to insert.</td>
</tr>
<tr>
<td><code>categoryvalue</code></td>
<td>The category for this data value on the category axis. The datatype of <code>categoryvalue</code> should match the datatype of the category axis. In most cases, you should include <code>categoryvalue</code>. Otherwise, an uncategorized value will be added to the series. (optional)</td>
</tr>
</tbody>
</table>

**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 does not 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:

```powerscript
gr_data.InsertData(1, 0, 44, "Qty")
```

is equivalent to:

```powerscript
gr_data.ModifyData(1, 0, 44, "Qty")
```

and is also equivalent to:

```powerscript
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:

```powerscript
gr_data.InsertData(1, 4, 44, "Qty")
```

- **ModifyData** changes the category label and the data value at position 4:

```powerscript
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`:

```powerscript
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:

```powerscript
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
rename.InsertDocument ( filename, clearflag { , filetype } )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rename</td>
<td>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.</td>
</tr>
<tr>
<td>filename</td>
<td>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.</td>
</tr>
<tr>
<td>clearflag</td>
<td>A boolean value specifying whether the new file will replace the current contents of the control. Values are:</td>
</tr>
<tr>
<td></td>
<td>• true – Replace the current contents with the file</td>
</tr>
<tr>
<td></td>
<td>• false – Insert the file into the existing contents at the insertion point</td>
</tr>
<tr>
<td>filetype</td>
<td>A value of the FileType enumerated datatype specifying the type of file being opened. Values are:</td>
</tr>
<tr>
<td>(optional)</td>
<td>• FileTypeRichText! – (Default) The file being opened is in rich text format (RTF)</td>
</tr>
<tr>
<td></td>
<td>• FileTypeText! – The file being opened is plain ASCII text (TXT)</td>
</tr>
<tr>
<td></td>
<td>• FileTypeHTML! – The file being opened is in HTML format (HTM or HTML)</td>
</tr>
<tr>
<td></td>
<td>• FileTypeDoc! – The file being opened is in Microsoft Word format (DOC)</td>
</tr>
</tbody>
</table>

If filetype is not specified, PowerBuilder uses the filename extension to decide whether to read the file as rich text or plain text. If the extension is not one of the supported file type extensions, PowerBuilder attempts to read the file as rich text. To insert files with extensions such as INI, LOG, or SQL, you must specify FileTypeText!.
InsertDocument

Description

Inserts a document into a rich text control when the control’s DisplayOnly property is set to true. If you try to do this, PowerBuilder displays a runtime error message.

When the control supports headers and footers (the HeaderFooter property is set to true), inserting a document can replace, but not add to, existing header and footer text. You must set clearflag to true to replace the existing header and footer text with header and footer text from the inserted document.

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

Any unsupported formatting is ignored.

Examples

This example inserts a document into rte_1 and reports the return value in a StaticText control:

```powerbuilder
integer rtn
rtn = rte_1.InsertDocument("c:\pb\test.rtf", &
TRUE, FileTypeRichText!)
sth_status.Text = String(rtn)
```

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.

Syntax

```powerbuilder
olecontrol.InsertFile ( filename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olecontrol</td>
<td>The name of the OLE control.</td>
</tr>
<tr>
<td>filename</td>
<td>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.</td>
</tr>
</tbody>
</table>
InsertItem

Return value

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

-1 File not found
-9 Other error

Usage

The contents of the specified file is embedded in the OLE object. There is no further link between the object in PowerBuilder and the file.

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.

<table>
<thead>
<tr>
<th>To insert an item into a</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListBox or DropDownListBox control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>PictureListBox or DropDownPictureListBox control</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>ListView control when only the label and picture index need to be specified</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>ListView control when more than the label and picture index need to be specified</td>
<td>Syntax 4</td>
</tr>
<tr>
<td>TreeView control when only the label and picture index need to be specified</td>
<td>Syntax 5</td>
</tr>
<tr>
<td>TreeView control when more than the label and picture index need to be specified</td>
<td>Syntax 6</td>
</tr>
</tbody>
</table>
**Syntax 1**

**For ListBox and DropDownListBox controls**

**Description**
Inserts an item into the list of values in a list box.

**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 at runtime.

**Examples**
This statement inserts the item Run Application before the fifth item in `lbactions`:

```powerscript
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:

```powerscript
string ls_City = "Denver"
integer li_pos
li_pos = lb_Cities.InsertItem(ls_City, 1)
```
### Syntax 2

**For ListBox and DropDownListBox controls**

**Description**
Inserts an item into the list of values in a picture list box.

**Applies to**
PictureListBox and DropDownPictureListBox controls

**Syntax**

```
listboxname.InsertItem ( item, pictureindex, index )
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listboxname</code></td>
<td>The name of the PictureListBox or DropDownPictureListBox into which you want to insert an item</td>
</tr>
<tr>
<td><code>item</code></td>
<td>A string whose value is the text of the item you want to insert</td>
</tr>
<tr>
<td><code>pictureindex</code> (optional)</td>
<td>An integer specifying the index of the picture you want to associate with the newly added item</td>
</tr>
<tr>
<td><code>index</code></td>
<td>The number of the item in the list before which you want to insert the item</td>
</tr>
</tbody>
</table>

**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 do not specify a picture index, the newly added item will not have a picture.

If you specify a picture index that does not 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 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.InsertItem("Run Application", 4, 5)
```
Syntax 3  
For ListView controls
Description
Inserts an item into a ListView control.
Applies to
ListView controls
Syntax
`listviewname.InsertItem(index, label, pictureindex)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listviewname</code></td>
<td>The name of the ListView control to which you are adding an item</td>
</tr>
<tr>
<td><code>index</code></td>
<td>An integer whose value is the index number of the item before which you are inserting a new item</td>
</tr>
<tr>
<td><code>label</code></td>
<td>A string whose value is the name of the item you are adding</td>
</tr>
<tr>
<td><code>pictureindex</code></td>
<td>An integer whose value is the index number of the picture of the item you are adding</td>
</tr>
</tbody>
</table>

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:

```powerscript
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)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listviewname</code></td>
<td>The name of the ListView control into which you are inserting an item</td>
</tr>
<tr>
<td><code>index</code></td>
<td>An integer whose value is the index number of the item you are adding</td>
</tr>
</tbody>
</table>

See also
AddItem
### InsertItem

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>A system structure of datatype ListViewItem in which InsertItem stores the item you are inserting</td>
</tr>
</tbody>
</table>

**Return value** Integer. Returns index 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:

```powershell
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)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The name of the TreeView control in which you want to insert an item.</td>
</tr>
<tr>
<td>handleparent</td>
<td>The handle of the item one level above the item you want to insert. To insert an item at the first level, specify 0.</td>
</tr>
<tr>
<td>handleafter</td>
<td>The handle of the item on the same level that you will insert the item immediately after.</td>
</tr>
<tr>
<td>label</td>
<td>The label of the item you are inserting.</td>
</tr>
<tr>
<td>pictureindex</td>
<td>The Index of the index of the picture you are adding to the image list.</td>
</tr>
</tbody>
</table>

**Return value** Long. Returns the handle of the inserted item if it succeeds and -1 if an error occurs.
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 do not 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:

```powerscript
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The name of the TreeView control into which you want to insert an item.</td>
</tr>
<tr>
<td>handleparent</td>
<td>The handle of the item one level above the item you want to insert. To insert an item at the first level, specify 0.</td>
</tr>
<tr>
<td>handleafter</td>
<td>The handle of the item on the same level that you will insert the item immediately after.</td>
</tr>
<tr>
<td>item</td>
<td>A TreeViewItem structure for the item you are inserting.</td>
</tr>
</tbody>
</table>

**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.
**InsertItemFirst**

If you are inserting the first child of an item, use `InsertItemLast` or `InsertItemFirst` instead. Those functions do not 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:

```plaintext
long ll_tvi, ll_tvparent
treeviewitem l_tvi
ll_tvi = tv_list.FindItem(currenttreeitem!, 0)
ll_tvparent = tv_list.FindItem(parenttreeitem!, ll_tvi)
tv_list.GetItem(ll_tvi, l_tvi)
tv_list.InsertItem(ll_tvparent, ll_tvi, l_tvi)
```

**See also**

`GetItem`

---

**InsertItemFirst**

**Description**

Inserts an item as the first child of a parent item.

**To insert an item as the first child of its parent**

<table>
<thead>
<tr>
<th>Use</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you only need to specify the item label and picture index</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>When you need to specify more than the item label and picture index</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**Description**

Inserts an item as the first child of its parent.

**Applies to**

TreeView controls

**Syntax**

```
treeviewname.InsertItemFirst ( handleparent, label, pictureindex )
```

**Table**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The TreeView control in which you want to specify an item as the first child of its parent.</td>
</tr>
<tr>
<td>handleparent</td>
<td>The handle of the item that will be the inserted item’s parent. To insert the item at the first level, specify 0.</td>
</tr>
<tr>
<td>label</td>
<td>The label of the item you want to specify as the first child of its parent.</td>
</tr>
<tr>
<td>pictureindex</td>
<td>The picture index for the item you want to specify as the first child of its parent.</td>
</tr>
</tbody>
</table>
Return value Long. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.

Examples

This example populates the first level of a TreeView using InsertItemFirst:

```powerscript
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The TreeView control in which you want to specify an item as the first child of its parent.</td>
</tr>
<tr>
<td>handleparent</td>
<td>The handle of the item that will be the inserted item’s parent. To insert the item at the first level, specify 0.</td>
</tr>
<tr>
<td>item</td>
<td>A TreeViewItem structure for the item you are inserting.</td>
</tr>
</tbody>
</table>

Return value Long. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.
InsertItemLast

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:

```powershell
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
InsertItemFirst
InsertItemLast
InsertItemSort

InsertItemLast

Inserts an item as the last child of a parent item.

<table>
<thead>
<tr>
<th>To insert an item as the last child of its parent</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you only need to specify the item label and picture index</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>When you need to specify more than item label and picture index</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The TreeView control in which you want to specify an item as the last child of its parent.</td>
</tr>
<tr>
<td>handleparent</td>
<td>The handle of the item that will be the inserted item’s parent. To insert the item at the first level, specify 0.</td>
</tr>
</tbody>
</table>
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:

```powerscript
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
**InsertItemLast**

**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.

<table>
<thead>
<tr>
<th>To insert an item in sorted order</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you only need to specify the item label and picture index</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>When you need to specify more than the item label and picture index</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1

**For TreeView controls**

Inserts an item in sorted order, if possible.

**Description**

Inserts an item in sorted order, if possible.

**Applies to**

TreeView controls

**Syntax**

```
treeviewname.InsertItemSort ( handleparent, label, pictureindex )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>treeviewname</code></td>
<td>The TreeView control in which you want to insert and sort an item as a child of its parent, according to its label.</td>
</tr>
<tr>
<td><code>handleparent</code></td>
<td>The handle of the item that will be the inserted item’s parent. To insert the item at the first level, specify 0.</td>
</tr>
<tr>
<td><code>label</code></td>
<td>The label by which you want to sort the item as a child of its parent.</td>
</tr>
<tr>
<td><code>pictureindex</code></td>
<td>The picture index for the item you want to sort as a child of its parent, according to its label.</td>
</tr>
</tbody>
</table>

**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 do not 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:

```powerscript
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

See also
InsertItem
InsertItemLast
InsertItemFirst

Syntax 2
For TreeView controls
Inserts an item in sorted order, if possible.
Applies to TreeView controls
Syntax
					
treeviewname.InsertItemSort ( handleparent, item )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treviewname</td>
<td>The TreeView control in which you want to sort an item as a child of its parent, according to its label.</td>
</tr>
<tr>
<td>handleparent</td>
<td>The handle of the item that will be the inserted item’s parent. To insert the item at the first level, specify 0.</td>
</tr>
<tr>
<td>item</td>
<td>A TreeViewItem structure for the item you are inserting.</td>
</tr>
</tbody>
</table>

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 do not 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:

```powerScript
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.

**Syntax**

```
olecontrol.InsertObject()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olecontrol</td>
<td>The name of the OLE control in which you want to insert an object</td>
</tr>
</tbody>
</table>

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

- 1 User canceled out of dialog box
- 9 Error

If any argument’s value is null, InsertObject returns null.

**Examples**
This example displays the standard Insert Object dialog box 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 an image at the insertion point in a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
\[ \text{rtename.InsertPicture( } \text{filename}, \text{ format} \text{ )} \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control in which you want to insert an image</td>
</tr>
<tr>
<td>filename</td>
<td>A string whose value is the name of the file that contains the image</td>
</tr>
<tr>
<td>format</td>
<td>An integer whose value determines the format in which the image is saved in the RTF file. Values are:</td>
</tr>
<tr>
<td></td>
<td>• 1 – BMP</td>
</tr>
<tr>
<td></td>
<td>• 2 – WMF (default)</td>
</tr>
<tr>
<td></td>
<td>• 3 – PNG</td>
</tr>
<tr>
<td></td>
<td>• 4 – JPG</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if it succeeds. Returns -2 if the compression format of a .tif file is not supported and -1 if a different error occurs. If filename is null, InsertPicture returns null.

Usage
If there is a selection, InsertPicture inserts the image at the beginning of the selection. The image and the selection remain selected. The following file types are supported: .bmp, .wmf, .jpeg, .png, and .gif. The .tif file type is also supported if it uses the LZW compression format.

Examples
This example inserts a PNG file at the insertion point in the RichTextEdit control \( \text{rte}_1 \). The PNG file will be saved in the RTF file in WMF format:

```
integer li_rtn
li_rtn = rte_1.InsertPicture("c:\windows\earth.png")
```

This example inserts a JPG file at the insertion point in the RichTextEdit control \( \text{rte}_1 \). The JPG file will be saved in the RTF file in JPG format:

```
integer li_rtn
li_rtn = rte_1.InsertPicture("c:\windows\earth.jpg", 4)
```

See also
InputFieldInsert
InsertDocument
## 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**
```
ccontrolname.InsertSeries ( seriesname, seriesnumber )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the graph in which you want to insert a series.</td>
</tr>
<tr>
<td><code>seriesname</code></td>
<td>A string containing the name of the series you want to insert. The series name must be unique within the graph.</td>
</tr>
<tr>
<td><code>seriesnumber</code></td>
<td>The number of the series before which you want to insert the new series. To add the new series at the end, specify 0.</td>
</tr>
</tbody>
</table>

**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: \( \text{Int}(n) \)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>The number for which you want the largest whole number that is less than or equal to it</td>
</tr>
</tbody>
</table>

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, \( \text{Int} \) returns 0. If \( n \) is null, \( \text{Int} \) returns null.

Usage: When the result for \( \text{Int} \) would be smaller than \(-32768\) or larger than \(32767\), \( \text{Int} \) returns 0 because the result cannot be represented as an integer.

Examples: These statements return 3.0:

\[
\begin{align*}
\text{Int}(3.2) \\
\text{Int}(3.8)
\end{align*}
\]

The following statements return -4.0:

\[
\begin{align*}
\text{Int}(-3.2) \\
\text{Int}(-3.8)
\end{align*}
\]

These statements remove the decimal portion of the variable and store the resulting integer in \( \text{li_nbr} \):

\[
\begin{align*}
\text{integer li_nbr} \\
\text{li_nbr} &= \text{Int}(3.2) \quad \text{// li_nbr = 3}
\end{align*}
\]

See also: Ceiling, Round, Truncate

Int method for DataWindows in the DataWindow Reference or the online Help

---

**Integer**

Description: Converts the value of a string to an integer or obtains an integer value that is stored in a blob.

Syntax: \( \text{Integer}(\text{stringorblob}) \)
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 datatype. 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` method for DataWindows in the *DataWindow Reference* or the online Help
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 InternetResult standard class user object.

Applies to
InternetResult objects

Syntax
 servicedereference.InternetData ( data )

Argument | Description
--- | ---
 servicedereference | 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 InternetResult. 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 InternetResult 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:

```PowerBuilder
MessageBox("HTML from GetURL", &
String(data, EncodingANSI!))
RETURN 1
```

The blob contains the actual data and is not Unicode encoded, therefore you must use the EncodingANSI! argument of the String function.

See also
GetURL
PostURL
**IntHigh**

**Description**
Returns the high word of a long value.

**Syntax**

```
IntHigh ( long )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>A long value</td>
</tr>
</tbody>
</table>

**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:

```powerscript
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>A long value</td>
</tr>
</tbody>
</table>

**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:

```powerscript
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 } } )

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:

```javascript
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
_Is_A

Description
Checks to see whether a CORBA object is an instance of a class that implements a particular interface.

This function is used by PowerBuilder clients connecting to EAServer.

Applies to
CORBAObject objects

Syntax
`corbaobject._Is_A(classname)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>corbaobject</code></td>
<td>An object of type CORBAObject that you want to test</td>
</tr>
<tr>
<td><code>classname</code></td>
<td>The interface that will be used for the test</td>
</tr>
</tbody>
</table>

Return value
Boolean. Returns true if the class of the object implements the specified interface and false if it does not.

Usage
Before making a call to _Narrow, you can call _Is_A to verify that a CORBA object is an instance of a class that implements the interface to which you want to narrow the object.

Examples
The following example checks to see that a CORBA object reference is an instance of a class that implements n_Bank_Account:

```pascal
CORBAObject  my_corbaobj
n_Bank_Account  my_account
...
...
if (my_corbaobj._is_a("n_Bank_Account")) then
    my_corbaobj._narrow(my_account,"n_Bank_Account")
end if
my_account.withdraw(100.0)
```

See also
_Narrow
### IsAlive

**Description**
Determines whether a server object is still running.

**Applies to**
OLEObject objects, OLETxnObject objects

**Syntax**

```
oleobject.IsAlive()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>The name of an OLEObject or OLETxnObject variable that is connected to an automation server or COM object</td>
</tr>
</tbody>
</table>

**Return value**
Boolean. Returns true if the server object appears to be running and false if it is dead.

**Usage**
Use the `IsAlive` function to determine whether a server process has died. This function does not replace the error-handling capability provided by the `ExternalException` and `Error` events. It provides a way to check the viability of the server at intervals or before specific operations to avoid runtime errors.

If `IsAlive` returns true, the server may only appear to be running, because the true state of the server may be masked. This is more likely to occur when the server is running on a different computer, because DCOM may be using cached information to determine the state of the server. A false return value always indicates that the server is dead.

**Examples**
This example creates an OLEObject variable and calls `ConnectToNewObject` to create and connect to a new instance of a PowerBuilder COM object. After performing some processing, it checks whether the server is still running before performing additional processing:

```powerbuilder
OLETxnObject EmpObj
Integer li_rc

EmpObj = CREATE OLEObject
li_rc = EmpObj.ConnectToNewObject("PB70COM.employee")
// Perform some work with the COM object
...
IF EmpObj.IsAlive() THEN
// Continue processing
END IF
```
IsAllArabic

Description
Tests whether a particular string is composed entirely of Arabic characters.

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 a version of Windows that supports right-to-left languages, IsAllArabic is set to false.

Examples
Under a version of Windows that supports right-to-left languages, 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.

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 a version of Windows that supports right-to-left languages, IsAllHebrew is set to false.
Examples

Under a version of Windows that supports right-to-left languages, 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.

**Syntax**

```
IsAnyArabic ( string )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string whose value you want to test to find out if it contains at least one Arabic character</td>
</tr>
</tbody>
</table>

**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 a version of Windows that supports right-to-left languages, `IsAnyArabic` is set to false.

**Examples**

Under a version of Windows that supports right-to-left languages, 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.

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 a version of Windows that supports right-to-left languages, IsAnyHebrew is set to false.

Examples
Under a version of Windows that supports right-to-left languages, 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.

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 a version of Windows that supports right-to-left languages, IsArabic is set to false.
Examples

Under a version of Windows that supports right-to-left languages, this statement returns true if the SingleLineEdit `sle_name` begins with an Arabic character:

```
IsArabic(sle_name.Text)
```

See also

`IsAllArabic`
`IsAnyArabic`
`IsArabic`
`IsArabicAndNumbers`
`Reverse`

---

**IsArabicAndNumbers**

**Description**
Tests whether a particular string is composed entirely of Arabic characters or numbers.

**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 a version of Windows that supports right-to-left languages, `IsArabicAndNumbers` is set to false.

**Examples**
Under a version of Windows that supports right-to-left languages, 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`
**IsCallerInRole**

**Description**
Indicates whether the direct caller of a COM object running on COM+ is in a specified role (either individually or as part of a group).

**Applies to**
TransactionServer objects

**Syntax**
```
transactionserver.IsCallerInRole ( role )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transactionserver</td>
<td>Reference to the TransactionServer service instance</td>
</tr>
<tr>
<td>role</td>
<td>A string expression containing the name of a role</td>
</tr>
</tbody>
</table>

**Return value**
Boolean. Returns `true` if the direct caller is in the specified role and `false` if it is not.

**Usage**
In COM+, a role is a name that represents the set of access permissions for a specific user or group of users. For example, a component that provides access to a sales database might have different roles for managers and salespersons.

In your code, you use `IsCallerInRole` to determine whether the caller of the current method is associated with a specific role before you execute code that performs a task restricted to users in that role.

`IsCallerInRole` only determines whether the direct caller of the current method is in the specified role. The direct caller may be either a client process or a server process.

**Package must run in a dedicated server process**
To support role-checking, the COM+ package must be activated as a Server package, not a Library package. Server packages run in a dedicated server process. Library packages run in the creator's process and are used primarily for debugging.

`IsCallerInRole` only returns a meaningful value when security checking is enabled. Security checking can be enabled in the COM/COM+ Project wizard or the Project painter.
Examples

The following example shows a call to a function (f_checkrole) that takes the name of a role as an argument and returns an integer. In this example only managers can place orders with a value over $20,000:

```powerscript
integer rc
long ordervalue
IF ordervalue > 20,000 THEN
    rc = f_checkrole("Manager")
    IF rc <> 1
        // handle negative values and exit
    ELSE
        // continue processing
    END IF
END IF
```

The f_checkrole function checks whether a component is running on COM+ and if security checking is enabled. Then it checks whether the direct caller is in the role passed in as an argument. If any of the checks fail, the function returns a negative value:

```powerscript
TransactionServer ts
integer li_rc
string str_role
li_rc = GetContextService( "TransactionServer", ts)
// handle error if necessary

// Find out if running on COM+
IF ts.which() <> 2 THEN RETURN -1

// Find out if security is enabled
IF NOT ts.IsSecurityEnabled() THEN RETURN -2

// Find out if the caller is in the role
IF NOT ts.IsCallerInRole(str_role) THEN
    RETURN -3
ELSE
    RETURN 1
END IF
```

See also

- ImpersonateClient
- IsImpersonating
- IsSecurityEnabled
- RevertToSelf
**IsDate**

Description
Tests whether a string value is a valid date.

Syntax

```
IsDate ( datevalue )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datevalue</td>
<td>A string whose value you want to test to determine whether it is a valid date</td>
</tr>
</tbody>
</table>

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 datatype. To convert a value into a date value, use the Date function. The year value must be in the range 01 to 9999.

Examples
This statement returns true:

```
IsDate("Jan 1, 05")
```

This statement returns false:

```
IsDate("Jan 32, 2005")
```

If the SingleLineEdit `sle_Date_Of_Hire` contains 7/1/99, these statements store 1999-07-01 in `HireDate`:

```
Date HireDate
IF IsDate(sle_Date_Of_Hire.text) THEN
    HireDate = Date(sle_Date_Of_Hire.text)
END IF
```

See also
IsDate method for DataWindows in the DataWindow Reference or the online Help

**IsHebrew**

Description
Tests whether a particular character is a Hebrew character. For a string, IsHebrew tests only the first character on the left.

Syntax

```
IsHebrew ( character )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>A character or string whose value you want to test to find out if it is an Hebrew character</td>
</tr>
</tbody>
</table>
Return value
Boolean. Returns true if character is an Hebrew character and false if it is not.

Usage
If you are not running a version of Windows that supports right-to-left languages, IsHebrew is set to false.

Examples
Under a version of Windows that supports right-to-left languages, this statement returns true if the SingleLineEdit sle_name begins with a Hebrew character:

\[
\text{IsHebrew}(\text{sle\_name}.\text{Text})
\]

See also
IsAllHebrew
IsAnyHebrew
IsHebrewAndNumbers
Reverse

IsHebrewAndNumbers

Description
Tests whether a particular string is composed entirely of Hebrew characters and numbers.

Syntax
\[ \text{IsHebrewAndNumbers}\left(\text{string}\right) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string whose value you want to test to find out if it is composed entirely of Hebrew characters and numbers</td>
</tr>
</tbody>
</table>

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 a version of Windows that supports right-to-left languages, IsHebrewAndNumbers is set to false.

Examples
Under a version of Windows that supports right-to-left languages, this statement returns true if the SingleLineEdit sle_name is composed entirely of Hebrew characters and numbers:

\[
\text{IsHebrewAndNumbers}(\text{sle\_name}.\text{Text})
\]

See also
IsAllHebrew
IsAnyHebrew
IsHebrew
Reverse
IsImpersonating

Description
Queries whether a COM object running on COM+ is impersonating the client.

Applies to
TransactionServer objects

Syntax
transactionserver.IsImpersonating()

Return value
Boolean. Returns true if the component is impersonating the client and false if it is not.

Usage
COM objects running on COM+ can use the ImpersonateClient function to run in the client’s security context so that the server process has access to the same resources as the client. Use IsImpersonating to determine whether the ImpersonateClient function has been called without a matching call to RevertToSelf.

Examples
The following example creates an instance of the TransactionServer service and checks whether the COM object is currently running on the client’s security context. If it is not, it impersonates the client, performs some processing using the client’s security context, then reverts to the object’s security context:

TransactionServer txninfo_test
integer li_rc

li_rc = GetContextService( "TransactionServer", &txninfo_test )
IF NOT txninfo_test.IsImpersonating() THEN
   txninfo_test.ImpersonateClient()
END IF
// continue processing as client
txninfo_test.RevertToSelf()

See also
ImpersonateClient
IsCallerInRole
IsSecurityEnabled
RevertToSelf
### IsInTransaction

**Description**  
Indicates whether a component is executing in a transaction.

**Applies to**  
TransactionServer objects

**Syntax**  
`transactionserver.IsInTransaction()`

**Argument** | **Description**  
--- | ---  
`transactionserver` | Reference to the TransactionServer service instance

**Return value**  
Boolean. Returns `true` if the component is executing as part of a transaction and `false` if it is not.

**Usage**  
Component methods can call `IsInTransaction` to determine whether they are executing within a transaction.

Methods in components that are declared to be transactional always execute as part of a transaction.

Methods in components that have a transaction type of Supports Transaction may or may not be running in the context of an EAServer transaction, depending on whether the component is instantiated directly by a base client or by another component. In components that have this transaction type, you can use `IsInTransaction` to determine whether the component is running in a transaction.

The `IsInTransaction` function corresponds to the `isInTransaction` transaction primitive in EAServer.

**Examples**  
The following example shows the use of the `IsInTransaction` function:

```powerscript
TransactionServer ts
Integer li_rc
long ll_rv

li_rc = this.GetContextService("TransactionServer", & ts)
IF ts.IsInTransaction = TRUE THEN
    // execute logic based on the transaction context
END IF
```

**See also**  
EnableCommit  
IsTransactionAborted  
Lookup  
SetAbort  
SetComplete  
Which
**IsNull**

**Description**
Reports whether the value of a variable or expression is null.

**Syntax**
```
IsNull ( any )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>A variable or expression that you want to test to determine whether its value is null</td>
</tr>
</tbody>
</table>

**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.

If one or more columns in a DataWindow are required columns, that is, they must contain data, you do not 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. In standard PowerBuilder applications, if a variable is not set to null explicitly by calling the `SetNull` function, calling the `IsNull` function against the variable returns false.

In general, the same applies in .NET applications. However, if the variable is of a reference type (a type derived from the `PowerObject` base class), `IsNull` returns true if the variable has not been initialized by assigning an instantiated object to it. In the following example, `IsNull` returns false in a standard PowerBuilder application, but it returns true in a .NET application:

```powerbuilder
dataStore ds
boolean b

b = IsNull(ds)
MessageBox("IsNull", string(b))
```

If the variable is explicitly set to null, `IsNull` returns true in both standard and .NET applications:

```powerbuilder
SetNull(ds)
b = IsNull(ds)
MessageBox("IsNull", string(b))
```

To ensure consistent behavior in standard and .NET applications, use the `IsValid` function to check whether an object has been instantiated instead of using the `IsNull` function. In the following example, each of the calls to `IsValid` returns false in both standard and .NET applications:
dataStore ds
boolean b

b = IsValid(ds)
MessageBox("IsValid", string(b))

SetNull(ds)
b = IsValidl(ds)
MessageBox("IsValid", string(b))

Examples
These statements set lb_test to true:

integer a, b
boolean lb_test

SetNull(b)
lb_test = IsNull(a + b)

See also
IsValid
SetNull
IsNull method for DataWindows in the DataWindow Reference or the online Help

IsNumber

Description
Reports whether the value of a string is a number.

Syntax
IsNumber ( string )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string whose value you want to test to determine whether it is a valid PowerScript number</td>
</tr>
</tbody>
</table>

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 datatype, use the Double, Dec, Integer, Long, or Real function.

Examples
This statement returns true:

IsNumber("32.65")
This statement returns false:

```powerbuilder
IsNumber("A16")
```

If the SingleLineEdit `sle_Age` contains 32, these statements store 32 in `li_YearsOld`:

```powerbuilder
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 method for DataWindows in the *DataWindow Reference* or the online Help

---

### IsPreview

**Description**

Reports whether a RichTextEdit control is in preview mode.

**Applies to**

RichTextEdit controls

**Syntax**

```powerbuilder
rtename.IsPreview()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control for which you want to know whether it is in preview mode</td>
</tr>
</tbody>
</table>

**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 is not already in preview mode and then prints it:

```powerbuilder
IF NOT rte_1.IsPreview() THEN
   rte_1.Preview(TRUE)
   rte_1.Print(1, "1-4", FALSE, TRUE)
END IF
```

See also

- Preview
IsSecurityEnabled

Description: Indicates whether or not security checking is enabled for a COM object running on COM+.

Applies to: TransactionServer objects

Syntax: `transactionserver.IsSecurityEnabled()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transactionserver</code></td>
<td>Reference to the TransactionServer service instance</td>
</tr>
</tbody>
</table>

Return value: Boolean. Returns `true` if security checking is enabled and `false` if it is not.

Usage: Use `IsSecurityEnabled` to determine whether security checking is enabled for the current COM object.

If the COM object is running in the creator’s process, `IsSecurityEnabled` always returns `false`.

Examples: The following example determines whether security checking is enabled and, if it is, checks whether the direct caller is in the Manager role before completing the call:

```powershell
TransactionServer ts
integer li_rc
string str_role = "Admin"

li_rc = GetContextService( "TransactionServer", & ts )
// Find out if security is enabled.
IF ts.IsSecurityEnabled() THEN
    // Find out if the caller is in the role.
    IF NOT ts.IsCallerInRole(str_role) THEN
        // do not complete call
    ELSE
        // execute call normally
    END IF
ELSE
    // security is not enabled
    // do not complete call
END IF
```

See also: ImpersonateClient, IsCallerInRole, IsImpersonating, RevertToSelf
IsTime

Description
Reports whether the value of a string is a valid time value.

Syntax
IsTime ( timevalue )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timevalue</td>
<td>A string whose value you want to test to determine whether it is a valid time</td>
</tr>
</tbody>
</table>

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 method for DataWindows in the DataWindow Reference or the online Help
**IsTransactionAborted**

**Description**
Determines whether the current transaction, in which an EAServer component participates, has been aborted.

**Applies to**
TransactionServer objects

**Syntax**
`transactionserver.IsTransactionAborted()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transactionserver</td>
<td>Reference to the TransactionServer service instance</td>
</tr>
</tbody>
</table>

**Return value**
Boolean. Returns true if the current transaction has been aborted and false if it has not.

**Usage**
The `IsTransactionAborted` function allows a component to verify that the current transaction is still viable before performing updates to the database. The `IsTransactionAborted` function corresponds to the `isRollbackOnly` transaction primitive in EAServer.

**Examples**
The following example checks to see whether the transaction has been aborted. If it has not, it updates the database and calls `EnableCommit`. If it has been aborted, it calls `DisableCommit`.

```powerscript
// Instance variables: ids_datastore, ts
Integer li_rc
long ll_rv

li_rc = this.GetContextService("TransactionServer", ts)
IF li_rc <> 1 THEN
    // handle the error
END IF
...
IF ts.IsTransactionAborted() = FALSE THEN
    ll_rv = ids_datastore.Update()
    IF ll_rv = 1 THEN
        ts.EnableCommit()
    ELSE
        ts.DisableCommit()
    END IF
END IF
```

**See also**
EnableCommit
IsInTransaction
Lookup
SetAbort
SetComplete
Which
IsValid

Description
Determines whether an object variable is instantiated—whether its value is a valid object handle.

Syntax
IsValid (objectvariable)

Argument | Description
---|---
objectvariable | An object variable or a variable of type Any—typically a reference to an object that you are testing for validity

Return value
Boolean. Returns true if objectvariable is an instantiated object. Returns false if objectvariable is not an object, or if it is an object that is not instantiated. If objectvariable 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)
```

This example returns -1 because the IsValid function returns false. Although the objectvariable argument is a valid string, it is not an instantiated object. The IsValid method would return true only if la_value was an instantiated object:

```
any la_value

la_value = "I’m a string"
IF NOT IsValid(la_value) THEN return -1
```

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 datatype 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. See also the table of KeyCode values in Usage.

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.
### Table 10-5: KeyCode values for keyboard keys

<table>
<thead>
<tr>
<th>Type of key</th>
<th>KeyCode values and descriptions</th>
</tr>
</thead>
</table>
| 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! |
CHAPTER 10  PowerScript Functions

The following code checks whether the user pressed the F1 key or the Ctrl key and executes some statements appropriate to the key pressed:

```powerscript
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:

```powerscript
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):

```powerscript
IF KeyDown(65) THEN ...
```

This statement tests whether the user pressed the Shift key and the A key:

```powerscript
IF KeyDown(65) AND KeyDown(KeyShift!) THEN ...
```

This statement in a Clicked event checks whether the Shift is also pressed:

```powerscript
IF KeyDown(KeyShift!) THEN ...
```

---

### 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 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:

```powerscript
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:

```powerscript
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):

```powerscript
IF KeyDown(65) THEN ...
```

This statement tests whether the user pressed the Shift key and the A key:

```powerscript
IF KeyDown(65) AND KeyDown(KeyShift!) THEN ...
```

This statement in a Clicked event checks whether the Shift is also pressed:

```powerscript
IF KeyDown(KeyShift!) THEN ...
```
LastPos

Description
Finds the last position of a target string in a source string.

Syntax
```
LastPos ( string1, string2 {, searchlength } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td>The string in which you want to find string2.</td>
</tr>
<tr>
<td>string2</td>
<td>The string you want to find in string1.</td>
</tr>
<tr>
<td>searchlength (optional)</td>
<td>A long that limits the search to the leftmost searchlength characters of the source string string1. The default is the entire string.</td>
</tr>
</tbody>
</table>

Return value
Long. Returns a long whose value is the starting position of the last occurrence of string2 in string1 within the characters specified in searchlength. If string2 is not found in string1 or if searchlength is 0, LastPos returns 0. If any argument’s value is null, LastPos returns null.

Usage
The LastPos function is case sensitive. The entire target string must be found in the source string.

Examples
This statement returns 6, because the position of the last occurrence of RU is position 6:
```
LastPos("BABE RUTH", "RU")
```
This statement returns 3:
```
LastPos("BABE RUTH", "B")
```
This statement returns 0, because the case does not match:
```
LastPos("BABE RUTH", "be")
```
This statement searches the leftmost 4 characters and returns 0, because the only occurrence of RU is after position 4. The search length must be at least 7 (to include the complete string RU) before the statement returns 6 for the starting position of the last occurrence of RU:
```
LastPos("BABE RUTH", "RU", 4)
```
These statements change the text in the SingleLineEdit sle_group. The last instance of the text NY is changed to North East:
```
long place_nbr
place_nbr = LastPos(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 LastPos function finds the position of the (last) tab in the string and the Left and Mid functions extract the information to the left and right of the tab:

```power_script
string s, ls_left, ls_right
integer li_tab

s = dw_groups.GetBandAtPointer()
li_tab = LastPos(s, ";t")

ls_left = Left(s, li_tab - 1)
ls_right = Mid(s, li_tab + 1)
```

These statements tokenize a source string backwards:

```power_script
// Tokenize the source string backwards
// Results in "pbsyc115.dll  powerbuilder
// shared  sybase  programs  c:

string sSource = &c:\programs\sybase\shared\powerbuilder\pbsyc115.dll
string sFind = '\'
string sToken
long llStart, llEnd

llEnd = Len(sSource) + 1

DO

    llStart = LastPos(sSource, sFind, llEnd)
    sToken = Mid(sSource, (llStart + 1), & (llEnd - llStart))
    mle_comment.text += sToken + '  '
    llEnd = llStart - 1
LOOP WHILE llStart > 1
```

See also Pos
### Left

**Description**

Obtains a specified number of characters from the beginning of a string.

**Syntax**

```
Left (string, n)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string you want to search</td>
</tr>
<tr>
<td>n</td>
<td>A long specifying the number of characters you want to return</td>
</tr>
</tbody>
</table>

**Return value**

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` method for DataWindows in the *DataWindow Reference* or the online Help

### LeftA

**Description**

Temporarily converts a string from Unicode to DBCS based on the current locale, then returns the specified number of bytes from the string.

**Syntax**

```
LeftA (string, n)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string you want to search from left to right</td>
</tr>
<tr>
<td>n</td>
<td>A long specifying the number of bytes of the characters in the return string</td>
</tr>
</tbody>
</table>
Return value String. Returns the characters for the leftmost $n$ bytes in the source string if it succeeds and the empty string ("") if an error occurs. If any argument’s value is null, LeftA returns null. If $n$ is greater than or equal to the length of the string, LeftA returns the entire string. It does not add spaces to make the return value’s length equal to $n$.

Usage LeftA replaces the functionality that Left had in DBCS environments in PowerBuilder 9. In SBCS environments, Left, LeftW, and LeftA return the same results.

**LeftW**

Description Obtains a specified number of characters from the beginning of a string. This function is obsolete. It has the same behavior as Left in all environments.

Syntax \textbf{LeftW} ( \textit{string}, \textit{n} )

**LeftTrim**

Description Removes spaces from the beginning of a string.

Syntax \textbf{LeftTrim} ( \textit{string}, \textit{removeallspaces} )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{string}</td>
<td>The string you want returned with leading spaces deleted</td>
</tr>
<tr>
<td>\textit{removeallspaces}</td>
<td>A boolean indicating that all types of spaces should be deleted</td>
</tr>
</tbody>
</table>

Usage If you do not include the optional \textit{removeallspaces} argument or its value is false, only the space character (U+0020) is removed from the string.

If the \textit{removeallspaces} argument is set to true, all types of space characters are removed.

This is a list of white spaces:

- CHARACTER TABULATION (U+0009)
- LINE FEED (U+000A)
- LINE TABULATION (U+000B)
LeftTrimW

FORM FEED (U+000C)
CARRIAGE RETURN (U+000D)
SPACE (U+0020)
NO-BREAK SPACE (U+00A0)
EN QUAD (U+2000)
EM QUAD (U+2001)
EN SPACE (U+2002)
EM SPACE (U+2003)
THREE-PER-EM SPACE (U+2004)
FOUR-PER-EM SPACE (U+2005)
SIX-PER-EM SPACE (U+2006)
FIGURE SPACE (U+2007)
PUNCTUATION SPACE (U+2008)
THIN SPACE (U+2009)
HAIR SPACE (U+200A)
ZERO WIDTH SPACE (U+200B)
IDEOGRAPHIC SPACE (U+3000)
ZERO WIDTH NO-BREAK SPACE (U+FEFF)

Examples

This statement returns RUTH when the leading spaces are all space characters:

LeftTrim(" RUTH")

This statement returns RUTH when the leading spaces include other types of space characters such as tab characters:

LeftTrim(" RUTH", true)

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 method for DataWindows in the DataWindow Reference or the online Help

LeftTrimW

Description

Removes spaces from the beginning of a string. This function is obsolete. It has the same behavior as LeftTrim in all environments.

Syntax

LeftTrimW ( string)
Len

Description
Reports the length of a string or a blob.

Syntax
Len (stringorblob)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringorblob</td>
<td>The string or blob for which you want the length in number of characters or in number of bytes</td>
</tr>
</tbody>
</table>

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 do not specify a size for the blob, Len initially reports the blob’s length as 0. PowerBuilder assigns a size to the blob the first time you assign data to the blob. Len reports the length of the blob as the number characters it can contain.

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 22
In the second example, \textit{ib\_blob} is initialized to the size 100 when it is declared. When you call \textit{Len} for \textit{ib\_blob}, it always returns 100. This example uses \textbf{BlobEdit}, instead of \textbf{Blob}, to assign data to the blob because its size is already established. The declaration of the instance variable is:

\begin{verbatim}
  blob{100} ib\_blob
\end{verbatim}

The sample code is:

\begin{verbatim}
  long ll\_len
  ll\_len = \textit{Len}(ib\_blob) // ll\_len set to 100
  BlobEdit(ib\_blob, 1, "Test String")
  ll\_len = \textit{Len}(ib\_blob) // ll\_len set to 100
\end{verbatim}

\textbf{LenA}

\textbf{Description} When the argument is a string, temporarily converts the string from Unicode to DBCS based on the current locale, then calculates its length in bytes. When the argument is a blob, no conversion takes place.

\textbf{Syntax} \begin{verbatim}
  LenA (stringorblob)
\end{verbatim}

\textbf{Argument} \begin{verbatim}
  stringorblob
\end{verbatim} The string or blob for which you want the length in number of bytes

\textbf{Return value} Long. Returns a long whose value is the length of \texttt{stringorblob} if it succeeds and -1 if an error occurs. If \texttt{stringorblob} is null, \textit{Len} returns null.

\textbf{Usage} \textit{LenA} replaces the functionality that \textit{Len} had in DBCS environments in PowerBuilder 9.

In SBCS environments, \textit{Len}, \textit{LenW}, and \textit{LenA} return the same results. If you specify a size when you declare a blob, that is the size reported by \textit{LenA}. If you do not specify a size for the blob, \textit{LenA} initially reports the blob’s length as 0. PowerBuilder assigns a size to the blob the first time you assign data to the blob. \textit{LenA} reports the length of the blob as the number of single-byte characters it can contain. \textit{Len} and \textit{LenW} report the size of the blob as the number of double-byte characters it can contain.
CHAPTER 10  PowerScript Functions

LenW
Description  Reports the length of a string or a blob. This function is obsolete. It has the same behavior as Len in all environments.
Syntax  \texttt{LenW ( stringorblob )}

Length
Description  Reports the length in bytes of an open OLE stream.

\textbf{Len function}
To get the length of a string or blob, use the \texttt{Len} function.

Applies to  OLEStream objects
Syntax  \texttt{olestream.Length ( sizevar )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{olestream}</td>
<td>The name of an OLE stream variable that has been opened</td>
</tr>
<tr>
<td>\texttt{sizevar}</td>
<td>A long variable in which Length will store the size of \texttt{olestream}</td>
</tr>
</tbody>
</table>

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 \texttt{MYSTUFF.OLE} and assigns it to the OLEStorage object \texttt{stg_stuff}. Then it opens the stream called info in \texttt{stg_stuff} and assigns it to the stream object \texttt{olestr_info}. Finally, it finds out the stream’s length and stores the value in the variable \texttt{info_len}.

The example does not check the function’s return values for success, but you should be sure to check the return values in your code:

\begin{verbatim}
boolean lb_memexists
OLEStorage stg_stuff
OLEStream olestr_info
long info_len

stg_stuff = CREATE oleStorage
\end{verbatim}
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 } )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraryname</td>
<td>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.</td>
</tr>
<tr>
<td>comments</td>
<td>(optional) A string whose value is the comments you want to associate with the library.</td>
</tr>
</tbody>
</table>

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 do not specify an extension, LibraryCreate adds the extension .PBL.

Examples
This statement in 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")
```

See also
LibraryDelete
LibraryDirectory
LibraryExport
LibraryImport
CHAPTER 10  PowerScript Functions

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 datatype 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")
```

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 )
LibraryDirectory

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraryname</td>
<td>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.</td>
</tr>
<tr>
<td>objecttype</td>
<td>A value of the LibDirType enumerated datatype identifying the type of objects you want listed:</td>
</tr>
<tr>
<td></td>
<td>• DirAll! – All objects</td>
</tr>
<tr>
<td></td>
<td>• DirApplication! – Application objects</td>
</tr>
<tr>
<td></td>
<td>• DirDataWindow! – DataWindow objects</td>
</tr>
<tr>
<td></td>
<td>• DirFunction! – Function objects</td>
</tr>
<tr>
<td></td>
<td>• DirMenu! – Menu objects</td>
</tr>
<tr>
<td></td>
<td>• DirPipeline! – Pipeline objects</td>
</tr>
<tr>
<td></td>
<td>• DirProject! – Project objects</td>
</tr>
<tr>
<td></td>
<td>• DirQuery! – Query objects</td>
</tr>
<tr>
<td></td>
<td>• DirStructure! – Structure objects</td>
</tr>
<tr>
<td></td>
<td>• DirUserObject! – User objects</td>
</tr>
<tr>
<td></td>
<td>• DirWindow! – Window objects</td>
</tr>
</tbody>
</table>

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:
If you call LibraryDirectory with a PBD file as the first argument, no comments are displayed because they are not included in PBD files.

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 should contain three string columns. The columns must be wide enough to fit the data in the input string. If not, PowerBuilder reports validation errors.

To return the object’s type, use LibraryDirectoryEx.

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:

```powerbuilder
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)

See also
ImportString
LibraryCreate
LibraryDelete
LibraryDirectoryEx
LibraryExport
LibraryImport

LibraryDirectoryEx

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, any comments for the object, and the object’s type. You can get a list of all objects or just objects of a specified type.

Syntax

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraryname</td>
<td>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.</td>
</tr>
<tr>
<td>objecttype</td>
<td></td>
</tr>
</tbody>
</table>
LibraryDirectoryEx

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| `objectype` | A value of the LibDirType enumerated datatype 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. LibraryDirectoryEx returns a tab-separated list with one object per line. The format of the list is:

```
name ~t date/time modified ~t comments ~t type~n
```

Returns the empty string ("") if an error occurs. If any argument’s value is null, LibraryDirectoryEx returns null.

Usage:

If you call LibraryDirectoryEx with a PBD file as the first argument, no comments are displayed because they are not included in PBD files.

You can display the result of LibraryDirectoryEx in a DataWindow control by passing the returned string to the ImportString function for that DataWindow. The DataWindow should contain four string columns. The columns must be wide enough to fit the data in the input string. If not, PowerBuilder reports validation errors.

If you do not need to return the object’s type, you can use LibraryDirectory.

For an example of parsing tab-delimited data, see the Pos or LastPos function.

Examples:

This code imports the string returned by LibraryDirectoryEx to the DataWindow `dw_list` and then redraws the `dw_list`. The DataWindow was defined with an external source and four string columns:

```
String ls_entries

ls_entries = LibraryDirectoryEx( &
    "c:\pb\dwTemp.pbl", DirUserObject!)
dw_list.SetRedraw(FALSE)
```
dw_list.Reset()
dw_list.ImportString(ls_Entries)
dw_list.SetRedraw(TRUE)

See also
ImportString
LibraryCreate
LibraryDelete
LibraryDirectory
LibraryExport
LibraryImport

LibraryExport

Description
Exports an object from a library. The object is exported as syntax.

Syntax

```
LibraryExport ( libraryname, objectname, objecttype )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraryname</td>
<td>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.</td>
</tr>
<tr>
<td>objectname</td>
<td>A string whose value is the name of the object you want to export.</td>
</tr>
<tr>
<td>objecttype</td>
<td>A value of the LibExportType enumerated datatype identifying the type of objects you want to export:</td>
</tr>
<tr>
<td></td>
<td>• ExportApplication! – Application object</td>
</tr>
<tr>
<td></td>
<td>• ExportDataWindow! – DataWindow object</td>
</tr>
<tr>
<td></td>
<td>• ExportFunction! – Function object</td>
</tr>
<tr>
<td></td>
<td>• ExportMenu! – Menu object</td>
</tr>
<tr>
<td></td>
<td>• ExportPipeline! – Pipeline objects</td>
</tr>
<tr>
<td></td>
<td>• ExportProject! – Project objects</td>
</tr>
<tr>
<td></td>
<td>• ExportQuery! – Query objects</td>
</tr>
<tr>
<td></td>
<td>• ExportStructure! – Structure object</td>
</tr>
<tr>
<td></td>
<td>• ExportUserObject! – User objects</td>
</tr>
<tr>
<td></td>
<td>• ExportWindow! – Window object</td>
</tr>
</tbody>
</table>

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.
LibraryImport

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:

```powershell
String ls_dwsyn, ls_errors
ls_dwsyn = LibraryExport("c:\pb\dwTemp.pbl", & "d_emp", ExportDataWindow!)
dw_1.Create(ls_dwsyn, ls_errors)
```

See also
Create method for DataWindows in the DataWindow Reference or online Help
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 } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraryname</td>
<td>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.</td>
</tr>
<tr>
<td>objectname</td>
<td>A string specifying the name of the DataWindow object you want to import.</td>
</tr>
<tr>
<td>objecttype</td>
<td>A value of the LibImportType enumerated datatype identifying the type of object you want to import. The only supported object type is ImportDataWindow!.</td>
</tr>
<tr>
<td>syntax</td>
<td>A string specifying the syntax of the DataWindow object you want to import.</td>
</tr>
<tr>
<td>errors</td>
<td>A string variable that you want to fill with any error messages that occur.</td>
</tr>
<tr>
<td>comments (optional)</td>
<td>A string specifying the comments you want to associate with the entry.</td>
</tr>
</tbody>
</table>
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 at runtime, 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:

```powerscript
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:

```powerscript
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")
```

See also

Describe method for DataWindows in the DataWindow Reference or the online Help
LibraryCreate
LibraryDelete
LibraryDirectory
LibraryImport
LineCount

Determines the number of lines in an edit control that allows multiple lines.

Applies to
RichTextEdit, MultiLineEdit, EditMask, and DataWindow controls

Syntax

`editname.LineCount()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>editname</code></td>
<td>The name of the DataWindow control, EditMask, MultiLineEdit, or RichTextEdit for which you want the number of lines</td>
</tr>
</tbody>
</table>

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 datatype 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:

```powerbuilder
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`:

```powerbuilder
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()```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>editname</code></td>
<td>The name of the RichTextEdit, MultiLineEdit, or EditMask in which you want to determine the length of the line containing the insertion point</td>
</tr>
</tbody>
</table>

**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_CContact` and line 5 contains the text Select All, `il_linelength` is set to 10 (the length of line 5):
```
integer il_linelength
il_linelength = mle_CContact.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

\[ \text{instancename}.\text{LineList}( \text{list} ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{instancename}</td>
<td>Instance name of the ProfileRoutine object.</td>
</tr>
<tr>
<td>\text{list}</td>
<td>An unbounded array variable of datatype ProfileLine in which LineList stores a ProfileLine object for each line in the routine. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

Return value
ErrorReturn. Returns one of the following values:

- Success! – The function succeeded
- ModelNotExistError! – 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:

```powerbuilder
Long ll_cnt
ProfileLine lproln_line[]

lpro_model.BuildModel()
lpro_model.RoutineList(iprot_list)

FOR ll_cnt = 1 TO UpperBound(iprot_list)
   iprot_list[ll_cnt].LineList(lproln_line)
   ...
NEXT
```

See also
BuildModel
**LinkTo**

Establishes a link between an OLE control and a file or an item within the file.

**Syntax**

```
olecontrol.LinkTo( filename, sourceitem )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olecontrol</code></td>
<td>The name of the OLE control in which you want to insert a linked object.</td>
</tr>
<tr>
<td><code>filename</code></td>
<td>A string whose value is the file name containing the data that you want to insert in <code>olecontrol</code>, with a link connecting the object in PowerBuilder to the original data. If you do not specify <code>sourceitem</code>, a link is established with the whole file.</td>
</tr>
<tr>
<td><code>sourceitem</code></td>
<td>(optional) A string that names an item within file name to which you want to link. The way you specify <code>sourceitem</code> is determined by the OLE server application.</td>
</tr>
</tbody>
</table>

**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`:

```powerbuilder
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:

```powerbuilder
integer result
result = ole_1.LinkTo("c:\xls\expense.xls", & "R1C1:R5C5")
```

**See also**

InsertFile
InsertObject
PasteLink
PasteSpecial
**LoadInk**

**Description**
Loads ink from a file or blob into an InkPicture control.

**Applies to**
InkPicture controls

**Syntax**

```plaintext
inkpicname.LoadInk ( t | b )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inkpicname</td>
<td>The name of the InkPicture control into which you want to load ink.</td>
</tr>
<tr>
<td>t</td>
<td>A string containing the name and location of a file that contains the ink you want to load into the control.</td>
</tr>
<tr>
<td>b</td>
<td>The name of a blob passed by reference that contains the ink you want to load into the control.</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 1 for success and -1 for failure.

**Usage**
Use the `LoadInk` function to load ink that has been saved to a file or a blob into the control.

**Examples**
The following example loads ink from a file. Since the user will select a single file, the second argument to `GetFileOpenName` contains the file's path and its name, so the third argument can be ignored:

```plaintext
string ls_inkpath, ls_inkname
GetFileOpenName("Select Ink File", ls_inkpath, & ls_inkname)
ip_1.LoadInk(ls_inkpath)
```

The following example loads ink from a blob:

```plaintext
string ls_inkpath, ls_inkname
integer li_filenum
blob lblb_ink

GetFileOpenName("Select Ink File", ls_inkpath, & ls_inkname)
li_filenum = FileOpen(ls_inkpath, StreamMode!)
If li_filenum <> 1 Then
  FileRead(li_filenum, lblb_ink)
  FileClose(li_filenum)
  ip_1.LoadInk(lblb_ink)
End If
```

**See also**
LoadPicture
ResetInk
ResetPicture
SaveInk
Save
LoadPicture

Description
Loads a picture from a file or blob into an InkPicture control.

Applies to
InkPicture controls

Syntax
`inkpicname.LoadPicture ( t | b )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inkpicname</code></td>
<td>The name of the InkPicture control into which you want to load a picture.</td>
</tr>
<tr>
<td><code>t</code></td>
<td>A string containing the name and location of a file that contains the picture you want to load into the control.</td>
</tr>
<tr>
<td><code>b</code></td>
<td>The name of a blob passed by reference that contains the picture you want to load into the control.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 for success and -1 for failure.

Usage
Use the LoadPicture function to load an image into an InkPicture control.

Examples
The following example loads an image from a file. Since the user will select a single file, the second argument to GetFileOpenName contains the file’s path and its name, so the third argument can be ignored:

```powerscript
string ls_path, ls_name
GetFileOpenName("Select Image", ls_path, ls_name)
ip_1.LoadPicture(ls_path)
```

The following example loads an image from a blob:

```powerscript
string ls_path, ls_name
integer li_filenum
blob lblb_ink

GetFileOpenName("Select Ink File", ls_path, ls_name)
li_filenum = FileOpen(ls_path, StreamMode!)
If li_filenum <> 1 Then
    FileRead(li_filenum, lblb_ink)
    FileClose(li_filenum)
    ip_1.LoadInk(lblb_ink)
End If
```

See also
LoadInk
ResetInk
ResetPicture
SaveInk
Save
**Log**

Returns the natural logarithm of a number. For an ErrorLogging object, this function can be used to write a string to the log file maintained by the object’s container.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determine the natural logarithm of a number</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Write a string to a log file</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For all objects**

Description

Determines the natural logarithm of a number.

Syntax

\[ \text{Log}(n) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>The number for which you want the natural logarithm (base ( e )). The value of ( n ) must be greater than 0.</td>
</tr>
</tbody>
</table>

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:

\[ \text{Log}(10) \]

This statement returns –.693147. . .:

\[ \text{Log}(0.5) \]

Both these statements result in an error at runtime:

\[ \text{Log}(0) \]

\[ \text{Log}(-2) \]

After the following statements execute, the value of \( a \) is 200:

\[
\text{double } a, b = \text{Log}(200)
\]

\[
a = \text{Exp}(b) /// a = 200
\]

**See also**

Exp

LogTen

Log method for DataWindows in the DataWindow Reference or the online Help
**Syntax 2**

For ErrorLogging objects

Description
Writes a string to the log file maintained by the object’s container.

Applies to
ErrorLogging objects

Syntax
```
errorlogobj.Log ( message )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorlogobj</td>
<td>Reference to the ErrorLogging service instance</td>
</tr>
<tr>
<td>message</td>
<td>The text string you want to write to the log</td>
</tr>
</tbody>
</table>

Return value
None.

Usage
The ErrorLogging object provides the ability to write messages to the log file used by the object’s container, such as `jaguar.log` for EAServer.

Before you call the Log function, create an instance of the ErrorLogging service by calling the GetContextService function.

Examples
The following example shows how to write a string to the log for EAServer or COM+:

```
ErrorLogging el
this.GetContextService("ErrorLogging", el)
el.log("Write this string to log")
```

See also
GetContextService

---

**LogTen**

Description
Determines the base 10 logarithm of a number.

Syntax
```
LogTen ( n )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The number for which you want the base 10 logarithm. The value of n must not be negative.</td>
</tr>
</tbody>
</table>

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 of $\text{LogTen}(n)$. To obtain the value of $n$ in the equation $r = \text{LogTen}(n)$, 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 method for DataWindows in the DataWindow Reference or the online Help

Long

Converts data into data of type long. There are two syntaxes.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine two unsigned integers into a long value</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Convert a string whose value is a number into a long or to obtain a long value stored in a blob</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

For combining integers

Description
Combines two unsigned integers into a long value.

Syntax

`Long (lowword, highword)`
### Chapter 10: PowerScript Functions

**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:

```powerbuilder
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)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stringorblob</code></td>
<td>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. <code>Stringorblob</code> can also be an Any variable containing a string or blob.</td>
</tr>
</tbody>
</table>

**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 datatype. 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.
### LongLong

**Examples**

This statement returns 2167899876 as a long:

```powershell
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:

```powershell
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, LongLong, Real, Long method for DataWindows in the *DataWindow Reference* or the online Help.

### LongLong

Converts data into data of type longlong. There are two syntaxes.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine two unsigned long values into a longlong value</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Convert a string whose value is a number into a longlong or obtain a longlong value stored in a blob</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

#### Syntax 1

**For combining longs**

**Description**

Combines two unsigned longs into a longlong value.

**Syntax**

```powershell
LongLong (lowword, highword)
```

**Argument** | **Description**
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lowword</code></td>
<td>An UnsignedLong to be the low word in the longlong</td>
</tr>
<tr>
<td><code>highword</code></td>
<td>An UnsignedLong to be the high word in the longlong</td>
</tr>
</tbody>
</table>

**Return value**

LongLong. Returns the longlong if it succeeds and -1 if an error occurs. If any argument’s value is null, LongLong returns null.

**Usage**

Use LongLong for passing values to external C++ and Java functions.
Examples

These statements convert the UnsignedLongs $lLow$ and $lHigh$ into a long value:

```
UnsignedLong lLow //Low long 32 bits
UnsignedLong lHigh //High long 32 bits
longlong LLValue //LongLong value 64 bits
```

$lLow = 1234567890$
$lHigh = 9876543210$

$LLValue = \texttt{LongLong}(lLow, lHigh)$
$\texttt{MessageBox("LongLong Value", LLValue)}$

### Syntax 2 For converting strings and blobs

**Description**

Converts a string whose value is a number into a longlong or obtains a longlong value stored in a blob.

**Syntax**

```
LongLong ( stringorblob )
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringorblob</td>
<td>The string you want returned as a longlong or a blob in which the first value is the longlong value. The rest of the contents of the blob is ignored. Stringorblob can also be an Any variable containing a string or blob.</td>
</tr>
</tbody>
</table>

**Return value**

LongLong. Returns the value of stringorblob as a longlong if it succeeds and 0 if stringorblob is not a valid PowerScript number or if it is an incompatible datatype. 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 LongLong function.

**Examples**

This statement returns 216789987654321 as a longlong:

```
LongLong("216789987654321")
```

After assigning blob data from the database to $lb_blob$, the following example obtains the longlong value stored at position 20 in the blob:

```
longlong llb_num
llb_num = \texttt{LongLong(BlobMid(lb_blob, 20, 4))}
```

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

**See also**

Dec
Double
Integer
Real
Lookup

Allows a PowerBuilder client or component to obtain a factory or home interface in order to create an instance of an EAServer component. This function is used by PowerBuilder clients connecting to components running in EAServer, and by PowerBuilder components connecting to other components running on the same server.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtain the factory interface of a CORBA-compliant component running in EAServer</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Obtain the home interface of an EJB component running in EAServer</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

**For CORBA-compliant EAServer components**

**Description**

Allows a PowerBuilder client or component to obtain the factory interface of an EAServer component in order to create an instance of the component.

**Applies to**

Connection objects, TransactionServer objects

**Syntax**

`objname.Lookup (objectvariable, componentname)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objname</code></td>
<td>The name of the Connection object used to establish the connection or of an instance of the TransactionServer context object.</td>
</tr>
<tr>
<td><code>objectvariable</code></td>
<td>A global, instance, or local variable of the factory interface type.</td>
</tr>
<tr>
<td><code>componentname</code></td>
<td>A string whose value is the name of the component instance to be created. You can optionally prepend a package name followed by a slash to the component name (for example, &quot;mypackage/mycomponent&quot;).</td>
</tr>
</tbody>
</table>

**Return value**

Long. Returns 0 if it succeeds and a negative number if an error occurs.

**Usage**

The Lookup function can be used as an alternative to the CreateInstance function. It obtains a reference to a factory interface that you can use to create an instance of a component running in EAServer.

Use the Connection object’s Lookup function to enable a PowerBuilder client to access a component running in EAServer. You can supply a server name or a list of server names in the location property of the Connection object.
Use the TransactionServer object’s Lookup function to enable a PowerBuilder component running in EAServer to access another component running on the same server.

To use the Lookup function, you need to create an EAServer proxy library for the SessionManager package to obtain a proxy for the factory interface. Include this proxy library in your library list.

Examples

The following example uses Lookup to instantiate the factory interface for the n_Bank_Account component, then it uses the factory’s create method to create an instance of the component:

```powerscript
// Instance variable:
// Connection myconnect
Factory my_Factory
CORBAObject mycorbaobj
n_Bank_Account my_account
long ll_result

ll_result = & myconnect.lookup(my_Factory,"Bank/n_Bank_Account")
mycorbaobj = my_Factory.create()
mycorbaobj._narrow(my_account, "Bank/n_Bank_Account")
my_account.withdraw(100.0)
```

See also

CreateInstance

Syntax 2

For instances of an EJB component

Description

Allows a PowerBuilder client or component to obtain the home interface of an EJB component in EAServer in order to create an instance of the component.

Applies to

Connection objects, TransactionServer objects

Syntax

```
objname.Lookup (objectvariable , componentname {, homeid})
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objname</td>
<td>The name of the Connection object used to establish the connection or of an instance of the TransactionServer context object.</td>
</tr>
<tr>
<td>objectvariable</td>
<td>A global, instance, or local variable of the type of the home interface to be created.</td>
</tr>
</tbody>
</table>
Lookup

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>componentname</td>
<td>A string whose value is the name of the EJB component to be created. You can optionally prepend a package name followed by a slash to the component name (for example, &quot;mypackage/mycomponent&quot;).</td>
</tr>
<tr>
<td>homeid</td>
<td>A string whose value is the name of the home interface to be created. This argument is optional</td>
</tr>
</tbody>
</table>

Return value
Long. Returns 0 if it succeeds and a negative number if an error occurs.

Usage
You can also use the Lookup method of the EJBConnection PowerBuilder extension object to create an instance of an EJB component running on any J2EE compliant application server. For more information, see Lookup in the PowerBuilder Extension Reference.

The Lookup function creates an instance of the home interface of an EJB component so that you can use it to create an instance of the EJB. Use the Connection object’s Lookup function to enable a PowerBuilder client to access a component running in EAServer. You can supply a server name or a list of server names in the location property of the Connection object. Use the TransactionServer object’s Lookup function to enable a PowerBuilder component running in EAServer to access an EJB component running on the same server.

The Lookup function uses the standard CORBA naming service to resolve componentname to a CORBA object that is then narrowed to the home interface name of the component. If you do not specify the third argument to the Lookup function, PowerBuilder expects the home interface name to have the format PackageName/CompNameHome. However, most EJB components use a standard Java package directory structure and the home interface name has a format such as com/domain/project/CompNameHome.

You can ensure that a PowerBuilder client or component can locate the component’s home interface by supplying the third argument to the Lookup function to specify the home interface name. A component’s home interface name is defined in the com.sybase.jaguar.component.home.ids property in the EAServer repository. The home.ids property has a format like this:

IDL:com/domain/project/CompNameHome:1.0
The third argument should be the value of the component’s home.ids string without the leading IDL: and trailing :1.0. For example:

```powerscript
ts.lookup(MyCartHome, "shopping/cart", "com/sybase/shopping/CartHome")
```

Alternatively, you can use the fully-qualified Java class name of the home interface specified in dot notation. For example:

```powerscript
ts.lookup(MyCartHome, "shopping/cart", "com.sybase.shopping.CartHome")
```

**Lookup is case sensitive**

Lookup in EAServer is case sensitive. Make sure that the case in the string you specify in the argument to the lookup function matches the case in the `ejb.home` property.

**Examples**

The following example uses `Lookup` with the Connection object to locate the home interface of the Multiply session EJB in the Java package `abc.xyz.math`:

```powerscript
// Instance variable:
// Connection myconnect
Multiply myMultiply
MultiplyHome myMultiplyHome
long ll_result, ll_product

ll_result = &
    myconnect.lookup(myMultiplyHome, "Math/Multiply", "abc.xyz.math.MultiplyHome")
IF ll_result <> 0 THEN
    MessageBox("Lookup failed", myconnect.errtext)
ELSE
    try
        myMultiply = myMultiplyHome.create()
        catch (ctscomponents_createexception ce)
            MessageBox("Create exception", ce.getmessage())
            // handle exception
        end try
        ll_product = myMultiply.multiply(1234, 4567)
    END IF
```

Entity beans have a `findByPrimaryKey` method that you can use to find an EJB saved in the previous session. This example uses that method to find a shopping cart saved for Dirk Dent:

```powerscript
// Instance variable:
// Connection myconnect
```
Cart myCart
CartHome myCartHome
long ll_result

ll_result = &
  myconnect.lookup(myCartHome,"Shopping/Cart", &
  "com.mybiz.shopping.CartHome")
IF ll_result <> 0 THEN
  MessageBox("Lookup failed", myconnect.errtext)
ELSE
  TRY
    myCart = myCartHome.findByPrimaryKey("DirkDent")
    myCart.addItem(101)
  CATCH ( ctscomponents_finderexception fe )
    MessageBox("Finder exception", &
    fe.getmessage())
  END TRY
END IF

Nonvisual objects deployed from PowerBuilder to EAServer can use an instance of the TransactionServer context object to locate the home interface of an EJB component in the same server:

CalcHome MyCalcHome
Calc MyCalc
TransactionServer ts
ErrorLogging errlog
long ll_result

this.getContextService("TransactionServer", ts)
this.getContextService("ErrorLogging", errlog)
ll_result = ts.lookup(MyCalcHome, "Utilities/Calc", &
  "com.biz.access.utilities.CalcHome")
IF ll_result <> 0 THEN
  errlog.log("Lookup failed: " + string(ll_result))
ELSE
  TRY
    MyCalc = MyCalcHome.create()
    MyCalc.square(12)
  CATCH (ctscomponents_createexception ce)
    errlog.log("Create exception: " + ce.getmessage())
  END TRY
END IF

See also
ConnectToServer
CHAPTER 10  PowerScript Functions

Lower

Description  Converts all the characters in a string to lowercase.

Syntax  

\[
\text{Lower} \left( \text{string} \right)
\]

Return value  String. Returns \text{string} with uppercase letters changed to lowercase if it succeeds and the empty string (""") if an error occurs. If \text{string} is null, \text{Lower} returns null.

Examples  

This statement returns \text{babe ruth}:

\[
\text{Lower}("\text{Babe Ruth}")
\]

See also  Upper

Lower method for DataWindows in the \text{DataWindow Reference} or the online Help

LowerBound

Description  Obtains the lower bound of a dimension of an array.

Syntax  

\[
\text{LowerBound} \left( \text{array} \left[ , \text{n} \right] \right)
\]

Return value  Long. Returns the lower bound of dimension \text{n} of \text{array} and -1 if \text{n} is greater than the number of dimensions of the array. If any argument’s value is null, \text{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:

```powerbuilder
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.

**Applies to**

`mailSession` object

**Syntax**

```powerbuilder
mailSession.mailAddress ( { mailmessage } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mailsession</code></td>
<td>A <code>mailSession</code> object identifying the session in which you want to address the message.</td>
</tr>
<tr>
<td><code>mailmessage</code></td>
<td>A <code>mailMessage</code> structure containing information about the message. If you omit <code>mailmessage</code>, <code>mailAddress</code> displays an Address dialog box.</td>
</tr>
</tbody>
</table>

**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 do not 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 cannot 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:

```PowerScript
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
```
mailDeleteMessage

Description
Deletes a mail message from the user’s electronic mail inbox.

Applies to
mailSession object

Syntax

mailsession.mailDeleteMessage ( messageid )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailsession</td>
<td>A mailSession object identifying the session in which you want to delete the message</td>
</tr>
<tr>
<td>messageid</td>
<td>A string whose value is the ID of the mail message to be deleted</td>
</tr>
</tbody>
</table>

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:

```powerbuild
string sID
integer nRow
mailReturnCode mRet
```
CHAPTER 10 PowerScript Functions

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.

Applies to mailSession object

Syntax mailSession.mailGetMessages ( { messagetype, } { unreadonly } )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailsession</td>
<td>A mailSession object identifying the session in which you want to get the messages.</td>
</tr>
<tr>
<td>messagetype (optional)</td>
<td>A string whose value is a message type. The default message type is IPM or an empty string (&quot;&quot;), 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.</td>
</tr>
</tbody>
</table>
| 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.
**mailGetMessages**

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:

```powerbuilder
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`
CHAPTER 10  PowerScript Functions

mailHandle
Description  Obtains the handle of a mailSession object.
Applies to  mailSession object
Syntax  \texttt{mailsession.mailHandle()}  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailsession</td>
<td>A mailSession object identifying the session for which you want the handle</td>
</tr>
</tbody>
</table>

Return value  UnsignedLong. Returns the internal handle of the mail session object. If \texttt{mailsession} is null, \texttt{mailHandle} displays an error message.
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 does not implement directly.
Before calling mail functions, you must declare and create a mailSession object and call \texttt{mailLogon} to establish a mail session.

Examples  This statement returns the handle of the current mail session:

\begin{verbatim}
current_session. mailHandle()
\end{verbatim}

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.
Applies to  mailSession object
Syntax  \texttt{mailsession.mailLogoff()}  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailsession</td>
<td>A mailSession object identifying the session from which you want to log off</td>
</tr>
</tbody>
</table>

Return value  mailReturnCode. Returns one of the following values:
- \texttt{mailReturnSuccess}!
- \texttt{mailReturnFailure}!
- \texttt{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:

```cpp
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.

Applies to
mailSession object

Syntax
```cpp
mailSession.mailLogon ( { profile, password } {, logonoption } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailsession</td>
<td>A mailSession object identifying the session you want to logon to.</td>
</tr>
<tr>
<td>profile (optional)</td>
<td>A string whose value is the user’s mail system profile or user ID.</td>
</tr>
<tr>
<td>password (optional)</td>
<td>A string whose value is the user’s mail system password.</td>
</tr>
<tr>
<td>logonoption (optional)</td>
<td>A value of the mailLogonOption enumerated datatype specifying the logon options:</td>
</tr>
<tr>
<td></td>
<td>• mailNewSession! – Starts a new mail session, whether or not the mail application is already running</td>
</tr>
<tr>
<td></td>
<td>• 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</td>
</tr>
<tr>
<td></td>
<td>• mailNewSessionWithDownLoad! – Starts a new mail session and forces new messages to be downloaded from the server to the user’s inbox</td>
</tr>
</tbody>
</table>

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 do not direct MailLogon to start a new session and the mail application is already running on the user’s computer, then the PowerBuilder mail session attaches to the existing session. A profile and password are not necessary.

When MailLogon establishes a new session, then the mail system’s dialog box prompts for the profile and password if the script does not 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 profile and password:

```powerscript
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 will not be prompted for an ID and password because user information is provided. Here the mailSession object is a local variable:

```powerscript
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.

Applies to
mailSession object

Syntax
mailsession.mailReadMessage ( messageid, mailmessage, readoption, mark )

Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailsession</td>
<td>A mailSession object identifying the session in which you want to read a message.</td>
</tr>
<tr>
<td>messageid</td>
<td>A string whose value is the ID of the mail message you want to read.</td>
</tr>
<tr>
<td>mailmessage</td>
<td>A mailMessage structure in which mailReadMessage stores the message information.</td>
</tr>
</tbody>
</table>

readoption A value of the mailReadOption enumerated datatype:
- 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.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.
**Reading attachments**

If a message has an attachment and you do not 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.

---

**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:

```powerbuilder
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')

    // Reread 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 Code Examples sample application supplied with PowerBuilder.

**See also**

- `mailGetMessages`
- `mailLogon`
- `mailSend`
mailRecipientDetails

Description
Displays a dialog box with the specified recipient’s address information.

Applies to
mailSession object

Syntax
`mailsession.mailRecipientDetails ( mailrecipient {, allowupdates } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailsession</td>
<td>A mailSession identifying the session in which you want to display the detail information for a recipient.</td>
</tr>
<tr>
<td>mailrecipient</td>
<td>A mailRecipient structure containing valid address information. Mailrecipient must contain a recipient identifier returned by mailAddress, mailResolveRecipient, or mailReadMessage.</td>
</tr>
<tr>
<td>allowupdates</td>
<td>(optional) A boolean indicating whether updates to the recipient’s name will be allowed. If the user does not have update privileges for the mail system, then allowupdates is ignored. The default is false.</td>
</tr>
</tbody>
</table>

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:

```powerbuilder
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 e-mail 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.

**Applies to**

mailSession object

**Syntax**

`mailSession.mailResolveRecipient (recipient [, allowupdates] )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mailsession</code></td>
<td>A mailSession object identifying the session in which you want to resolve the recipient.</td>
</tr>
<tr>
<td><code>recipient</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>allowupdates</code> (optional)</td>
<td>A boolean indicating whether updates to the recipient’s name will be allowed. If the user does not have update privileges for the mail system, then <code>allowupdates</code> is ignored. The default is <code>false</code>.</td>
</tr>
</tbody>
</table>

**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 corrects the information if it differs from the mail system. If you set allowupdates to true and the information differs from the mail system, mailResolveRecipient corrects 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:

```powerbuilder
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
```

`PowerBuilder`
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:

```powerscript
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.

**Applies to**: `mailSession` object

**Syntax**

```
mailSession.mailSaveMessage ( messageid, mailmessage )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mailSession</code></td>
<td>A <code>mailSession</code> object identifying the session in which you want to save the mail message.</td>
</tr>
<tr>
<td><code>messageid</code></td>
<td>A string whose value is the message ID of the message being replaced. If you are saving a new message, specify an empty string (“”).</td>
</tr>
<tr>
<td><code>mailmessage</code></td>
<td>A <code>mailMessage</code> structure containing the message being saved.</td>
</tr>
</tbody>
</table>
mailSaveMessage

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:

```powerbuilder
mailRecipient recip
mailMessage msg
mailReturnCode mRet

recip.Name = "Smith, Jerry"
mRet = mSes.mailResolveRecipient(recip)
IF mRet <> mailReturnSuccess! THEN
    MessageBox("Save New Message", 
              "Invalid address.")
    RETURN
END IF

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.")
END IF
```
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:

```powerbuilder
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"
recip.Name = mSes.mailResolveRecipient( recip )
msg.Recipient[1] = recip
mRet = mSes.mailSaveMessage(s_ID, msg)
IF mRet <> mailReturnSuccess! THEN
    MessageBox("Save Old Message", "Failed somehow.")
END IF

See also

- mailReadMessage
- mailResolveRecipient
```

See also the mail examples in the samples that are supplied with PowerBuilder.
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.

Applies to
mailSession object

Syntax

```powerbuilder
mailSession.mailSend ( { mailMessage } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailSession</td>
<td>A mailSession object identifying the session in which you want to send the mail message</td>
</tr>
<tr>
<td>mailMessage</td>
<td>A mailMessage structure</td>
</tr>
</tbody>
</table>

Return value
mailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!
- mailReturnLoginFailure!
- 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.

For mailSend, mailOriginator! is not a valid value for the Recipient property of the mailMessage object. The valid values are mailto!, mailcc!, and mailbcc!. To specify that the sender receive a copy of the message, use mailcc!.

Examples
These statements create a mail session, send a message, and then log off the mail system and destroy the mail session object:

```powerbuilder
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string in which you want to look for a pattern of characters</td>
</tr>
<tr>
<td>textpattern</td>
<td>A string whose value is the text pattern</td>
</tr>
</tbody>
</table>

**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.
**Match**

**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.

**Table 10-6: Metacharacters used by Match function**

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caret (^)</td>
<td>Matches the beginning of a string</td>
<td>^C matches C at the beginning of a string.</td>
</tr>
<tr>
<td>Dollar sign ($)</td>
<td>Matches the end of a string</td>
<td>s$ matches s at the end of a string.</td>
</tr>
<tr>
<td>Period (.)</td>
<td>Matches any character</td>
<td>. . . matches three consecutive characters.</td>
</tr>
<tr>
<td>Backslash ()</td>
<td>Removes the following metacharacter’s special characteristics so that it matches itself</td>
<td>$ matches $.</td>
</tr>
<tr>
<td>Character class (a group of characters enclosed in square brackets ([ ]))</td>
<td>MATCHES ANY OF THE ENCLOSED CHARACTERS</td>
<td>[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.</td>
</tr>
<tr>
<td>Complemented character class (first character inside the brackets is a caret)</td>
<td>MATCHES ANY CHARACTER NOT IN THE GROUP FOLLOWING THE CARET</td>
<td>^[0-9] matches any character except a digit, and ^[A-Za-z] matches any character except a letter.</td>
</tr>
</tbody>
</table>
The metacharacters asterisk (*), plus (+), and question mark (?) are unary operators that are used to specify repetitions in a regular expression:

**Table 10-7: Unary operators used as metacharacters by Match function**

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>* (asterisk)</td>
<td>Indicates zero or more occurrences</td>
<td>A* matches zero or more As (no As, A, AA, AAA, and so on)</td>
</tr>
<tr>
<td>+ (plus)</td>
<td>Indicates one or more occurrences</td>
<td>A+ matches one A or more than one A (A, AAA, and so on)</td>
</tr>
<tr>
<td>? (question mark)</td>
<td>Indicates zero or one occurrence</td>
<td>A? matches an empty string (&quot;&quot;&quot;) or A</td>
</tr>
</tbody>
</table>

**Sample patterns** The following table shows various text patterns and sample text that matches each pattern:

**Table 10-8: Text pattern examples for Match function**

<table>
<thead>
<tr>
<th>This pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>Any string that contains AB; for example, ABA, DEABC, graphAB_one</td>
</tr>
<tr>
<td>B*</td>
<td>Any string that contains 0 or more Bs; for example, AC, B, BB, BBB, ABBBC, and so on</td>
</tr>
<tr>
<td>AB*C</td>
<td>Any string containing the pattern AC or ABC or ABBC, and so on (0 or more Bs)</td>
</tr>
<tr>
<td>AB+C</td>
<td>Any string containing the pattern ABC or ABBC or ABBBC, and so on (1 or more Bs)</td>
</tr>
<tr>
<td>ABB*C</td>
<td>Any string containing the pattern ABC or ABBC or ABBBC, and so on (1 B plus 0 or more Bs)</td>
</tr>
<tr>
<td>^AB</td>
<td>Any string starting with AB</td>
</tr>
<tr>
<td>AB?C</td>
<td>Any string containing the pattern AC or ABC (0 or 1 B)</td>
</tr>
<tr>
<td>^[ABC]</td>
<td>Any string starting with A, B, or C</td>
</tr>
<tr>
<td>^[ABC]</td>
<td>A string containing any characters other than A, B, or C</td>
</tr>
<tr>
<td>^[^abc]</td>
<td>A string that begins with any character except a, b, or c</td>
</tr>
<tr>
<td>^[^a-zA-Z]$</td>
<td>Any single-character string that is not a lowercase letter (^ and $ indicate the beginning and end of the string)</td>
</tr>
<tr>
<td>[A-Z]+</td>
<td>Any string with one or more uppercase letters</td>
</tr>
<tr>
<td>^[0-9]+$</td>
<td>Any string consisting only of digits</td>
</tr>
<tr>
<td>^[0-9][0-9][0-9]$</td>
<td>Any string consisting of exactly three digits</td>
</tr>
<tr>
<td>^([0-9][0-9][0-9])$</td>
<td>Any consisting of exactly three digits enclosed in parentheses</td>
</tr>
</tbody>
</table>
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):

\[
\text{Match}(\text{sle}_{\text{ID}}.\text{Text}, \ "^[A-Za-z]\")
\]

This statement returns false if the text in sle_ID contains any digits (^ inside a bracket is a complement operator):

\[
\text{Match}(\text{sle}_{\text{ID}}.\text{Text}, \ "[^0-9]\")
\]

This statement returns true if the text in sle_ID contains one uppercase letter:

\[
\text{Match}(\text{sle}_{\text{ID}}.\text{Text}, \ "[A-Z]\")
\]

This statement returns true if the text in sle_ID contains one or more uppercase letters (+ indicates one or more occurrences of the pattern):

\[
\text{Match}(\text{sle}_{\text{ID}}.\text{Text}, \ "[A-Z]+\")
\]

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):

\[
\text{Match}(\text{sle}_{\text{ID}}.\text{Text}, \ "^[0-9][0-9][A-Za-z]$\")
\]

See also Pos

Match method for DataWindows in the DataWindow Reference or the online Help

**MatchW**

**Description**

Determines whether a string’s value contains a particular pattern of characters. This function is obsolete. It has the same behavior as Match in all environments.

**Syntax**

\[
\text{MatchW} ( \text{string, textpattern} )
\]
Max

Description
Determines the larger of two numbers.

Syntax
\texttt{Max( x, y )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>The number to which you want to compare ( y )</td>
</tr>
<tr>
<td>( y )</td>
<td>The number to which you want to compare ( x )</td>
</tr>
</tbody>
</table>

Return value
The datatype of \( x \) or \( y \), whichever datatype is more precise. If any argument’s value is null, \texttt{Max} returns null.

Usage
If either of the values being compared is null, \texttt{Max} returns null.

Examples
This statement returns 7:
\texttt{Max(4, 7)}

This statement returns -4:
\texttt{Max(-4, -7)}

This statement returns 8.2, a decimal value:
\texttt{Max(8.2, 4)}

See also
Min
Max method for DataWindows in the \textit{DataWindow Reference} or the online Help
**MemberDelete**

**Description**
Deletes a member from an OLE object in a storage. The member can be another OLE object (a substorage) or a stream.

**Applies to**
OLEStorage objects

**Syntax**

```powerbuilder
olestorage.MemberDelete ( membername )
```

**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:

```powerbuilder
boolean lb_memexists
integer result

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

stg_stuff.MemberExists("wordobj", lb_memexists)
IF lb_memexists THEN
    result = stg_stuff.MemberDelete("wordobj")
    IF result = 0 THEN stg_stuff.Save()
END IF
```

**See also**
MemberExists
MemberRename
Open
MemberExists

Description
Determines whether the named member is part of an OLE object in a storage. The member can be another OLE object (a substorage) or a stream.

Applies to
OLEStorage objects

Syntax

\texttt{olestorage.MemberExists\ (\ membername, \ exists\ )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{olestorage}</td>
<td>The name of an object variable of type OLEStorage that you want to check</td>
</tr>
<tr>
<td>\texttt{membername}</td>
<td>A string whose value is the name of the member that you want to check</td>
</tr>
<tr>
<td>\texttt{exists}</td>
<td>A boolean variable that will store whether or not the member exists</td>
</tr>
</tbody>
</table>

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

-\texttt{1} The storage is not open
-\texttt{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 \texttt{wordobj} is a substorage within that object and, if so, deletes it and saves the object back to the file:

```
boolean lb_memexists
integer result

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

stg_stuff.MemberExists("wordobj", lb_memexists)
IF lb_memexists THEN
    result = stg_stuff.MemberDelete("wordobj")
    IF result = 0 THEN stg_stuff.Save( )
END IF
```

See also
MemberDelete
MemberRename
Open
MemberRename

Description

Renames a member in an OLE storage. The member can be another OLE object (a substorage) or a stream.

Applies to

OLEStorage objects

Syntax

`olestorage.MemberRename (membername, newname)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olestorage</code></td>
<td>The name of an object variable of type OLEStorage containing the member (substorage or stream) you want to rename</td>
</tr>
<tr>
<td><code>membername</code></td>
<td>A string whose value is the name of the member you want to rename</td>
</tr>
<tr>
<td><code>newname</code></td>
<td>A string whose value is the new name to be assigned to the member</td>
</tr>
</tbody>
</table>

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:

```powershell
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**
```powerbuilder
MessageBox ( title, text {, icon {, button {, default} } } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>A string specifying the title of the message box, which appears in the box's title bar.</td>
</tr>
<tr>
<td>text</td>
<td>The text you want to display in the message box. The text can be a numeric datatype, a string, or a boolean value.</td>
</tr>
</tbody>
</table>
| icon (optional) | A value of the Icon enumerated datatype 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 datatype 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 datatype. 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 a version of Windows that supports right-to-left languages 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 does not work**
Controls capture the mouse in order to perform certain operations. For instance, CommandButtons capture the mouse during mouse clicks, Edit controls capture for text selection, and scroll bars 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):

```powershell
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):

```powershell
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

Obtains a specified number of characters from a specified position in a string.

Syntax

```
Mid ( string, start {, length } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string from which you want characters returned.</td>
</tr>
<tr>
<td>start</td>
<td>A long specifying the position of the first character you want returned. (The position of the first character of the string is 1).</td>
</tr>
<tr>
<td>length</td>
<td>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. (optional)</td>
</tr>
</tbody>
</table>

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 ls_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 character 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:

```powerbuilder
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:

```powerbuilder
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 AscA
Left
Pos
Right
UpperBound
Mid method for DataWindows in the DataWindow Reference or the online Help

**MidA**

**Description**
Temporarily converts a string to DBCS, then returns the specified number of bytes from the string, starting from a specified position.

**Syntax**

```
MidA (string, start, {length})
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string you want to search.</td>
</tr>
<tr>
<td>start</td>
<td>A long specifying the position of the first byte you want returned. (The position of the first byte of the string is 1.)</td>
</tr>
<tr>
<td>length</td>
<td>A long whose value is the number of bytes you want returned. If you do not enter length or if length is greater than the number of bytes to the right of start, MidA returns the remaining bytes in the string.</td>
</tr>
</tbody>
</table>

**Return value**
String. Returns characters specified by the number of bytes searched in a source string, beginning at the byte specified in the start argument. If start is greater than the number of bytes in string, the MidA function returns an empty string ("""). If length is greater than the number of bytes remaining after the start character, MidA returns the remaining bytes. The return string is not filled with spaces to make it the specified length. If any argument’s value is null, MidA returns null.

**Usage**
MidA replaces the functionality that Mid had in DBCS environments in PowerBuilder 9.

**MidW**

**Description**
Obtains a specified number of characters from a specified position in a string. This function is obsolete. It has the same behavior as Mid.

**Syntax**

```
MidW (string, start, length)
```
**Min**

**Description**
Determines the smaller of two numbers.

**Syntax**

\[
\text{Min}(x, y)
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>The number to which you want to compare (y)</td>
</tr>
<tr>
<td>(y)</td>
<td>The number to which you want to compare (x)</td>
</tr>
</tbody>
</table>

**Return value**
The datatype of \(x\) or \(y\), whichever datatype is more precise. If any argument’s value is \(null\), \(\text{Min}\) returns \(null\).

**Usage**
If either of the values being compared is \(null\), \(\text{Min}\) returns \(null\).

**Examples**
This statement returns 4:

\[
\text{Min}(4, 7)
\]

This statement returns -7:

\[
\text{Min}(-4, -7)
\]

This statement returns 3.0, a decimal value:

\[
\text{Min}(9.2, 3.0)
\]

**See also**
Max

\(\text{Min}\) method for DataWindows in the *DataWindow Reference* or the online Help

---

**Minute**

**Description**
Obtains the number of minutes in the minutes portion of a time value.

**Syntax**

\[
\text{Minute}(\text{time})
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{time})</td>
<td>The time value from which you want the minutes</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns the minutes portion of \(\text{time}\) (00 to 59). If \(\text{time}\) is \(null\), Minute returns \(null\).

**Examples**
This statement returns 1:

\[
\text{Minute}(19:01:31)
\]

**See also**
Hour
Second

Minute method for DataWindows in the *DataWindow Reference* or online Help
**Mod**

**Description**
Obtains the remainder (modulus) of a division operation.

**Syntax**
```
Mod ( x, y )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>The number you want to divide by y</td>
</tr>
<tr>
<td>y</td>
<td>The number you want to divide into x</td>
</tr>
</tbody>
</table>

**Return value**
The datatype of x or y, whichever datatype 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 method for DataWindows in the *DataWindow Reference* or the online Help

---

**ModifyData**

Changes the value of a data point in a series on a graph. There are two syntaxes depending on the type of graph.

<table>
<thead>
<tr>
<th>To modify a data point in</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>All graph types except scatter</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Scatter graphs</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**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 (their data comes directly from the DataWindow).
ModifyData

Syntax

controlname.ModifyData (seriesnumber, datapoint, datavalue
{, categoryvalue })

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to modify data.</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number of the series in which you want to modify data.</td>
</tr>
<tr>
<td>datapoint</td>
<td>The number of the data point for which you want to modify the data.</td>
</tr>
<tr>
<td>datavalue</td>
<td>The new value of the data point. The datatype of datavalue is the same as the datatype of the values axis of the graph.</td>
</tr>
<tr>
<td>categoryvalue</td>
<td>The category for datavalue. The datatype of categoryvalue is the same as the datatype of the category axis of the graph. (optional)</td>
</tr>
</tbody>
</table>

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 a comparison of AddData, InsertData, and ModifyData, see the Usage section in InsertData.

Examples

These statements change the data for Apr in the series named Costs in the graph gr_product_data:

```powerbuilder
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 (their data comes directly from the DataWindow).
CHAPTER 10  PowerScript Functions

Syntax

```
controlname.ModifyData( seriesnumber, datapoint, xvalue, yvalue )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the scatter graph in which you want to modify data in a series</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number that identifies the series in which you want to modify data</td>
</tr>
<tr>
<td>datapoint</td>
<td>The number of the data point for which you want to modify data</td>
</tr>
<tr>
<td>xvalue</td>
<td>The new x value of the data you want to modify</td>
</tr>
<tr>
<td>yvalue</td>
<td>The new y value of the data you want to modify</td>
</tr>
</tbody>
</table>

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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>The date from which you want the month</td>
</tr>
</tbody>
</table>

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(2004-01-31)
```
Move

These statements store in $start\_month$ the month entered in the SingleLineEdit sle_start_date:

```powershell
integer start_month
start_month = Month(date(sle_start_date.Text))
```

See also
- Day
- Date
- Year
- Month method for DataWindows in the DataWindow Reference or the online Help

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

Syntax
```powershell
objectname.Move (x, y)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectname</td>
<td>The name of the object or control you want to move to a new location</td>
</tr>
<tr>
<td>x</td>
<td>The x coordinate of the new location in PowerBuilder units</td>
</tr>
<tr>
<td>y</td>
<td>The y coordinate of the new location in PowerBuilder units</td>
</tr>
</tbody>
</table>

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. For a line control, $x$ and $y$ are the BeginX and BeginY properties.

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, pop-up, 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 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:

\[
\begin{align*}
objectname.X &= x \\
objectname.Y &= y
\end{align*}
\]

These statements cause PowerBuilder to redraw *gb_box1* twice:

\[
\begin{align*}
gb\_box1.X &= 150 \\
gb\_box1.Y &= 200
\end{align*}
\]

This statement has the same result but redraws *gb_box1* once:

\[
gb\_box1.\text{Move}(150, 200)
\]

**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:

\[
gb\_box1.\text{Move}(150, 200)
\]

This statement moves the picture *p_Train2* next to the picture *p_Train1*:

\[
P\_\text{Train2.} \text{Move}(P\_\text{Train1.X + P\_Train1.Width, \& P\_Train1.Y})
\]

**MoveTab**

**Description** Moves a tab page to another position in a Tab control, changing its index number.

**Applies to** Tab controls

**Syntax**

\[
tab\text{controlname.} \text{MoveTab}(\text{source, destination})
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabcontrolname</td>
<td>The name of the Tab control containing the tab you want to move.</td>
</tr>
<tr>
<td>source</td>
<td>An integer whose value is the index of the tab you want to move.</td>
</tr>
</tbody>
</table>

PowerScript Reference 761
_Narrow

**Description**
Converts a CORBA object reference from a general supertype to a more specific subtype.

This function is used by PowerBuilder clients connecting to EAServer.

**Applies to**
CORBAObject objects

**Syntax**
```
corbaobject _Narrow ( newremoteobject, classname )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>corbaobject</td>
<td>An object of type CORBAObject that you want to convert</td>
</tr>
<tr>
<td>newremoteobject</td>
<td>A variable that will contain the converted object reference</td>
</tr>
<tr>
<td>classname</td>
<td>The class name of the subtype to which you want to narrow the object reference</td>
</tr>
</tbody>
</table>

**Return value**
Long. Returns 0 if it succeeds and a negative number if an error occurs.

---

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

---

moveTab

**Description**
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)
```

---

Narrow

**Description**
Converts a CORBA object reference from a general supertype to a more specific subtype.

This function is used by PowerBuilder clients connecting to EAServer.

**Applies to**
CORBAObject objects

**Syntax**
```
corbaobject _Narrow ( newremoteobject, classname )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination</td>
<td>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.</td>
</tr>
</tbody>
</table>

---

corbaobject

**Description**
Converts a CORBA object reference from a general supertype to a more specific subtype.

This function is used by PowerBuilder clients connecting to EAServer.

**Applies to**
CORBAObject objects

**Syntax**
```
corbaobject _Narrow ( newremoteobject, classname )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>corbaobject</td>
<td>An object of type CORBAObject that you want to convert</td>
</tr>
<tr>
<td>newremoteobject</td>
<td>A variable that will contain the converted object reference</td>
</tr>
<tr>
<td>classname</td>
<td>The class name of the subtype to which you want to narrow the object reference</td>
</tr>
</tbody>
</table>

**Return value**
Long. Returns 0 if it succeeds and a negative number if an error occurs.
CHAPTER 10  PowerScript Functions

Usage

The _Narrow function allows you to narrow proxy objects in a CORBA-compatible client that connects to EAServer. For additional examples, see the functions on the See also list.

Examples

The following example narrows a CORBA object reference to the n_Bank_Account interface:

```powerscript
CORBAObject my_corbaobj
n_Bank_Account my_account
...
...
my_corbaobj.\_narrow(my_account,"Bank/n_Bank_Account")

my_account.withdraw(100.0)
```

In this example, the component is an EJB component that resides in a separate domain in EAServer. In this case, the SimpleBean component’s classes are in the ../classes/adomain/asimplepackage subdirectory:

```powerscript
CORBAObject my_corbaobj
SimpleBean my_simplebean
SimpleBeanHome my_simplebeanhome
...
...
my_corbaobj.\_narrow(my_simplebeanhome,
  "adomain/asimplepackage/SimpleBeanHome")
```

See also

_Is_A
Resolve_Initial_References
String_To_Object

---

**NextActivity**

Description

Provides the next activity in a trace file.

Applies to

TraceFile objects

Syntax

```
instancename.\_NextActivity ()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instancename</td>
<td>Instance name of the TraceFile object</td>
</tr>
</tbody>
</table>

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:

```powershell
String ls_filename, ls_line
TraceFile ltf_file
TraceActivityNode ltan_node

ls_filename = s1e_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 method for DataWindows in the DataWindow Reference or the online Help

---

**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 )
```
ObjectAtPointer

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control for which you want the object under the pointer</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>An integer variable in which you want to store the number of the series under the pointer</td>
</tr>
<tr>
<td>datapoint</td>
<td>An integer variable in which you want to store the number of the data point under the pointer</td>
</tr>
</tbody>
</table>

Return value

grObjectType. Returns a value of the grObjectType enumerated datatype if the user clicks anywhere in the graph (including an empty area) and a null value if the user clicks outside the graph.

Values of grObjectType and the parts of the graph associated with them are:

- TypeCategory! – A label for a category
- TypeCategoryAxis! – The category axis or between the category labels
- TypeCategoryLabel! – The label of the category axis
- TypeData! – A data point or other data marker
- TypeGraph! – Any place within the graph control that is not 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 datatype 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, \texttt{ObjectAtPointer} stores the data point number in the designated variable.

When the user clicks in a series, but not on the actual data point, \texttt{ObjectAtPointer} stores 0 in \textit{datapoint} and when the user clicks in a category, \texttt{ObjectAtPointer} stores 0 in \textit{seriesnumber}. When the user clicks other parts of the graph, \texttt{ObjectAtPointer} stores 0 in both variables.

### Call \texttt{ObjectAtPointer} first

\texttt{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 \texttt{gr\_product} in \textit{SeriesNbr} and \textit{ItemNbr}. If the object type is TypeSeries! they obtain the series name, and if it is TypeData! they get the data value:

```powerscript
integer SeriesNbr, ItemNbr
double data_value
grObjectType object_type
string SeriesName

object_type = &
gr_product\_ObjectAtPointer\(\text{SeriesNbr}, \text{ItemNbr}\)

IF object_type = TypeSeries! THEN
    SeriesName = &
gr_product\_SeriesName\(\text{SeriesNbr}\)
ELSIF object_type = TypeData! THEN
    data_value = &
gr_product\_GetData\(\text{SeriesNbr}, \text{ItemNbr}\)
END IF
```

These statements store the series number and data point number at the pointer location in the graph named \texttt{gr\_computers} in the DataWindow control \texttt{dw\_equipment} in \textit{SeriesNbr} and \textit{ItemNbr}:

```powerscript
integer SeriesNbr, ItemNbr
dw\_equipment\_ObjectAtPointer\("gr\_computers", &
    SeriesNbr, ItemNbr)
Object_To_String

Description
Gets the string form of an object.
This function is used by PowerBuilder clients connecting to EAServer.

Applies to
JaguarORB objects

Syntax
\texttt{jaguarorb.\textbf{Object\_To\_String} ( \texttt{object} )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{jaguarorb}</td>
<td>An instance of JaguarORB.</td>
</tr>
<tr>
<td>\textit{object}</td>
<td>The CORBA object that will be converted to a string.</td>
</tr>
</tbody>
</table>

The string representation of a CORBA object is an Interoperable Object Reference (IOR) that describes how to connect to the server hosting the object. EAServer supports both standard format IORs (which are hex-encoded) and a URL format that is human-readable.

Return value
String. Returns the string representation of a CORBA object.

Usage
The \textbf{Object\_To\_String} function can be used to serialize a proxy object reference. By serializing an object reference, you can save the state of the object so that it persists after the client terminates processing.

\textbf{Object\_To\_String} is typically used in conjunction with \textbf{String\_To\_Object}, which allows you to deserialize an object reference.

Examples
The following example shows the use of the \textbf{Object\_To\_String} function to serialize a proxy object reference:

\begin{verbatim}
Payroll payroll
JaguarORB my_orb
...
my_orb = CREATE JaguarORB
my_orb.init("ORBRetryCount=3,ORBRetryDelay=1000")
...
String payroll_ior = my_orb.\textbf{Object\_To\_String}(payroll)
\end{verbatim}

See also
Init
\textbf{String\_To\_Object}
CHAPTER 10  PowerScript Functions

OffsetPos

Description
Sets the offset for progress bar controls.

Applies to
Progress bar controls

Syntax
`control.OffsetPos (increment)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>The name of the progress bar control</td>
</tr>
<tr>
<td>increment</td>
<td>An integer that is added to the start position of the progress bar control</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if it succeeds and -1 if there is an error.

Examples
This statement offsets the start position of a horizontal progress bar by 10:

```
HProgressBar.OffsetPos ( 10 )
```

See also
SelectionRange
SetRange
StepIt

Open

Opens a window, an OLE object, or a trace file.

For windows  Open displays a window and makes all its properties and controls available to scripts.

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.

<table>
<thead>
<tr>
<th>To open</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>An OLE object in a file and load it into an OLE control</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>An OLE object in a storage object in memory and load it into an OLE control</td>
<td>Syntax 4</td>
</tr>
</tbody>
</table>
Open

<table>
<thead>
<tr>
<th>Syntax 1</th>
<th>For windows of a known datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Opens a window object of a known datatype. Open displays the window and makes all its properties and controls available to scripts.</td>
</tr>
<tr>
<td>Applies to</td>
<td>Window objects</td>
</tr>
<tr>
<td>Syntax</td>
<td>Open ( windowvar {, parent } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowvar</td>
<td>The name of the window you want to display. You can specify a window object defined in the Window painter (which is a window datatype) or a variable of the desired window datatype. Open places a reference to the opened window in windowvar.</td>
</tr>
<tr>
<td>parent</td>
<td>The window you want make the parent of the child or pop-up window you are opening. If you open a child or pop-up window and omit parent, PowerBuilder associates the window being opened with the currently active window.</td>
</tr>
</tbody>
</table>

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.

For trace files Open opens the specified trace file for reading.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open a trace file</td>
<td>Syntax 8</td>
</tr>
</tbody>
</table>

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
**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 datatype, use Syntax 2 of Open.

**Parent windows for the opened window** Generally, if you are opening a child or a pop-up 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 pop-up 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 checks its parent, and that window’s parent, until it finds a window that it can use as a parent. Therefore if you open a pop-up window and specify a sheet as its parent, PowerBuilder makes the MDI frame that contains the sheet its parent.

If you do not specify a parent for a child or pop-up window, the active window becomes the parent. Therefore, if one pop-up is active and you open another pop-up, the first pop-up is the parent, not the main window. When the first pop-up is closed, PowerBuilder closes the second pop-up too.

However, 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.

**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 scroll bars 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`:

```powershell
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:

```powershell
child cw_data
Open(cw_data, w_employee)
```

The following code opens two windows of type `w_emp`:

```powershell
w_emp w_e1, w_e2
Open(w_e1)
Open(w_e2)
```

### Syntax 2

**For windows of unknown datatype**

**Description**

Opens a window object when you do not know its datatype until the application is running. `Open` displays the window and makes all its properties and controls available to scripts.

**Applies to**

Window objects

**Syntax**

```powershell
Open( windowvar, windowtype {, parent } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windowvar</code></td>
<td>A window variable, usually of datatype window. <code>Open</code> places a reference to the opened window in <code>windowvar</code>.</td>
</tr>
<tr>
<td><code>windowtype</code></td>
<td>A string whose value is the datatype of the window you want to open. The datatype of <code>windowtype</code> must be the same or a descendant of <code>windowvar</code>.</td>
</tr>
<tr>
<td><code>parent</code> (child and pop-up windows only) (optional)</td>
<td>The window you want to make the parent of the child or pop-up window you are opening. If you open a child or pop-up window and omit <code>parent</code>, PowerBuilder associates the window being opened with the currently active window.</td>
</tr>
</tbody>
</table>

**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 datatype as windowvar (the datatype includes datatypes inherited from it). The datatype 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 datatype. See the last example, in which the window datatypes 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 datatype 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 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:

```powerbuilder
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:

```powerbuilder
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`:

```powerbuilder
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.

**Applies to**  
OLE controls

**Syntax**  
`olecontrol.Open ( OLEsourcefile )`
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 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.

**Applies to**

OLE controls

**Syntax**

```
olecontrol.Open ( sourcestorage, substoragename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olecontrol</td>
<td>The name of the OLE control into which you want to load an OLE object</td>
</tr>
<tr>
<td>sourcestorage</td>
<td>The name of an object variable of OLEStorage containing the object you want to load into olecontrol</td>
</tr>
<tr>
<td>substoragename</td>
<td>A string specifying the name of a substorage that contains the desired object within storagename</td>
</tr>
</tbody>
</table>
Open

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.

Applies to

OLE storage objects

Syntax

```
OLEstorage.Open ( OLEsourcefile {, readmode {, sharemode } } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLEstorage</td>
<td>The name of an object variable of type OLEStorage into which you want to load the OLE object.</td>
</tr>
<tr>
<td>OLEsourcefile</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 wo not 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:

```powershell
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:

```powershell
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.

```powershell
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.

```powerscript
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.

**Applies to**
OLE storage objects

**Syntax**
```
olestorage.Open ( substoragename, readmode, sharemode, sourcestorage )
```
Open

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olestorage</td>
<td>The name of a object variable of type OLEStorage into which you want to load the OLE object.</td>
</tr>
<tr>
<td>substoragename</td>
<td>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.</td>
</tr>
<tr>
<td>readmode</td>
<td>A value of the enumerated datatype stgReadMode that specifies the type of access you want for substoragename. Values are:</td>
</tr>
<tr>
<td></td>
<td>• stgReadWrite! – Read/write access. If the member does not exist, Open creates it.</td>
</tr>
<tr>
<td></td>
<td>• stgRead! – Read-only access. You cannot change substoragename.</td>
</tr>
<tr>
<td></td>
<td>• stgWrite! – Write access. You can rewrite substoragename but not read its current contents. If the member does not exist, Open creates it.</td>
</tr>
<tr>
<td>sharemode</td>
<td>A value of the enumerated datatype stgShareMode that specifies how other attempts, by your own or other applications, to open substoragename will fare. Values are:</td>
</tr>
<tr>
<td></td>
<td>• stgExclusive! – (Default) No other attempt to open substoragename will succeed.</td>
</tr>
<tr>
<td></td>
<td>• stgDenyNone! – Any other attempt to open substoragename will succeed.</td>
</tr>
<tr>
<td></td>
<td>• stgDenyRead! – Other attempts to open substoragename for reading will fail.</td>
</tr>
<tr>
<td></td>
<td>• stgDenyWrite – Other attempts to open substoragename for writing will fail.</td>
</tr>
<tr>
<td>sourcestorage</td>
<td>An open OLEStorage object containing substoragename.</td>
</tr>
</tbody>
</table>

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 will not 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.

```powerbuilder
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**

**Description**
Opens a stream in an open OLE storage object and loads it into an OLE stream object.

**Applies to**
OLE stream objects

**Syntax**

```powershell
olestream.Open ( sourcestorage, streamname {, readmode {, sharemode } } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olestream</code></td>
<td>The name of a object variable of type OLEStream into which you want to load the OLE object.</td>
</tr>
<tr>
<td><code>sourcestorage</code></td>
<td>An OLE storage that contains the stream to be opened.</td>
</tr>
<tr>
<td><code>streamname</code></td>
<td>A string specifying the name of the stream within <code>sourcestorage</code> that you want to open.</td>
</tr>
<tr>
<td><code>readmode</code> (optional)</td>
<td>A value of the enumerated datatype stgReadMode that specifies the type of access you want for <code>streamname</code>. Values are:</td>
</tr>
<tr>
<td></td>
<td>• stgReadWrite! – Read/write access. If <code>streamname</code> does not exist, Open creates it.</td>
</tr>
<tr>
<td></td>
<td>• stgRead! – Read-only access. You cannot change <code>streamname</code>.</td>
</tr>
<tr>
<td></td>
<td>• stgWrite! – Write access. You can rewrite <code>streamname</code> but not read its current contents. If <code>streamname</code> does not exist, Open creates it.</td>
</tr>
<tr>
<td><code>sharemode</code> (optional)</td>
<td>A value of the enumerated datatype stgShareMode that specifies how other attempts, by your own or other applications, to open <code>streamname</code> will fare. Values are:</td>
</tr>
<tr>
<td></td>
<td>• stgExclusive! – No other attempt to open <code>streamname</code> will succeed.</td>
</tr>
<tr>
<td></td>
<td>• stgDenyNone! – Any other attempt to open <code>streamname</code> will succeed.</td>
</tr>
<tr>
<td></td>
<td>• stgDenyRead! – Other attempts to open <code>streamname</code> for reading will fail.</td>
</tr>
<tr>
<td></td>
<td>• stgDenyWrite – Other attempts to open <code>streamname</code> for writing will fail.</td>
</tr>
</tbody>
</table>
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
For opening trace files

Description
Opens the specified trace file for reading.

Applies to
TraceFile object

Syntax

\[ \text{instancename}.\text{Open} ( \text{filename} ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{instancename}</td>
<td>Instance name of the TraceFile object</td>
</tr>
<tr>
<td>\text{filename}</td>
<td>A string identifying the name of the trace file you want to read</td>
</tr>
</tbody>
</table>

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.\text{Open}(ls_filename)
...
```

See also
Close
Reset
NextActivity
OpenChannel

Description
Opens a channel to a DDE server application.

Syntax
OpenChannel ( applname, topicname {, windowhandle} )

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 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:

```powershell
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`:

```powershell
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**
```powerscript
OpenSheet ( sheetrefvar {, windowtype }, mdiframe {, position {, arrangeopen } } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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        | A string whose value is the datatype of the window you want to open. The datatype of `windowtype` must be the same or a descendant of `sheetrefvar`.
| mdiframe          | The name of an MDI frame window.                                            |
| position          | 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       | A value of the ArrangeOpen enumerated datatype 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. In some cases, such as if the `windowtype` argument is invalid, `OpenSheet` throws a runtime error and does not return a value; therefore, it is recommended that you both test the return value and wrap the function call in a try-catch block as shown in the first example in the Examples section.

**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 datatype as `sheetrefvar` (a datatype includes datatypes inherited from it). The datatype 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 occurs.

PowerBuilder does not 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 more information about PBR files for an executable, see the *PowerBuilder Users 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.

PowerBuilder cannot append the sheets to a menu that does not 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.

**Opening response windows**

*Do not* use the `OpenSheet` function to open a response window.
Examples

This example 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` might return -1 or throw a runtime error if the call fails. To ensure that both of these possibilities are trapped, this example checks the return value of the function and uses a try-catch statement to catch a possible runtime error:

```power_script
integer li_return
try
  li_return = OpenSheet (child_1, MDI_User, 2, & Original!)
  if IsNull(li_return) then
    MessageBox ("Failure", "Null argument provided")
  elseif li_return= 1 then
    MessageBox ("Success", "Sheet opened.")
  else
    MessageBox ("Failure", "Sheet open failed.")
  end if
catch (runtimeerror rt)
  MessageBox("Failure","Sheet open failed. " & + rt.getmessage()) //Handle the error or not
end try
```

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:

```power_script
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.
OpenSheetWithParm

Applies to Window objects

Syntax

```
OpenSheetWithParm ( sheetrefvar, parameter {, windowtype }, mdiframe {, position {, arrangeopen } } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sheetrefvar</td>
<td>The name of any window variable that is not an MDI frame window. OpenSheetWithParm places a reference to the open sheet in sheetrefvar.</td>
</tr>
<tr>
<td>parameter</td>
<td>The parameter you want to store in the Message object when the sheet is opened. Parameter must have one of these datatypes:</td>
</tr>
<tr>
<td></td>
<td>• String</td>
</tr>
<tr>
<td></td>
<td>• Numeric</td>
</tr>
<tr>
<td></td>
<td>• PowerObject</td>
</tr>
<tr>
<td>windowtype</td>
<td>(optional) A string whose value is the datatype of the window you want to open. The datatype of windowtype must be the same or a descendant of sheetrefvar.</td>
</tr>
<tr>
<td>mdiframe</td>
<td>The name of the MDI frame window in which you want to open this sheet.</td>
</tr>
<tr>
<td>position</td>
<td>(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.</td>
</tr>
<tr>
<td>arrangeopen</td>
<td>(optional) A value of the ArrangeOpen enumerated datatype specifying how you want the sheets arranged in the MDI frame when they are opened:</td>
</tr>
<tr>
<td></td>
<td>• Cascaded! – (Default) Cascade the sheet relative to other open sheets so that its title bar is below the previously opened sheet.</td>
</tr>
<tr>
<td></td>
<td>• Layered! – Layer the sheet so that it fills the frame and covers previously opened sheets.</td>
</tr>
<tr>
<td></td>
<td>• Original! – Open the sheet in its original size and cascade it.</td>
</tr>
</tbody>
</table>

Return value Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, OpenSheetWithParm returns null. In some cases, such as if the windowtype argument is invalid, OpenSheetWithParm throws a runtime error and does not return a value; therefore, it is recommended that you both test the return value and wrap the function call in a try-catch block as shown in the first example in the Examples section.

Usage The system Message object has three properties for storing data. Depending on the datatype of the parameter specified for OpenSheetWithParm, scripts for the opened sheet would check one of the following properties.
In the opened window, it is a good idea to access the value passed in the Message object immediately (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 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.

Opening response windows
Do not use the OpenSheetWithParm function to open a response window.

Examples

See the usage notes for OpenSheet, which also apply to OpenSheetWithParm.

This example 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 might return -1 or throw a runtime error if the call fails. To ensure that both of these possibilities are trapped, this example checks the return value of the function and uses a try-catch statement to catch a possible runtime error:

```powerbuilder
integer li_return
try
    li_return = OpenSheetWithParm(w_child_1, "MA", & MDI_User, 2, Original!)
    if IsNull(li_return) then
        MessageBox ("Failure", "Null argument provided")
    elseif li_return= 1 then
        MessageBox ("Success", "Sheet opened.")
    else
        MessageBox ("Failure", "Sheet open failed.")
    end if
```
catch (runtimeerror rt)
    Messagebox("Failure", "Sheet open failed. " &
            + rt.getmessage()) //Handle the error
end try

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!)
    end window

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.

<table>
<thead>
<tr>
<th>To open</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user object as a tab page</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>A user object as a tab page, allowing the application to select the user object’s type at runtime</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

For user objects of a known datatype

Description

Opens a custom visual user object of a known datatype as a tab page in a Tab control.

Applies to

Tab controls

Syntax

`tabcontrolname.OpenTab ( userobjectvar, index )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tabcontrolname</code></td>
<td>The name of the Tab control in which you want to open the user object as a tab page.</td>
</tr>
<tr>
<td><code>userobjectvar</code></td>
<td>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 datatype) or a variable of the desired user object datatype. <code>OpenTab</code> places a reference to the opened custom visual user object in <code>userobjectvar</code>.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>The number of the tab before which you want to insert the new tab. If <code>index</code> is 0 or greater than the number of tabs, the tab page is inserted at the end.</td>
</tr>
</tbody>
</table>

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 adds 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.

Behavior change
In previous releases, calling the OpenTab function to open a user object as a tab page displayed the tab page even if the user object’s Visible property was set to false. In current releases, the user object’s Visible property must be set to true for the tab page to display.

Examples
This statement opens an instance of a user object named u_Employee as a tab page in the Tab control tab_1:

```powershell
    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:

```powershell
    u_employee u_to_open
    tab_1.OpenTab(u_to_open, 1)
```

See also
OpenTabWithParm

Syntax 2
For user objects of unknown datatype

Description
Opens a visual user object as a tab page within a Tab control when the datatype of the user object is not known until the script is executed.

Applies to
Tab controls
Synta

tabcontrolname.OpenTab ( userobjectvar, userobjecttype, index )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabcontrolname</td>
<td>The name of the Tab control in which you want to open the user object as a tab page.</td>
</tr>
<tr>
<td>userobjectvar</td>
<td>A variable of datatype UserObject. OpenTab places a reference to the opened user object in userobjectvar.</td>
</tr>
<tr>
<td>userobjecttype</td>
<td>A string whose value is the name of the user object you want to open. The datatype of userobjecttype must be a descendant of userobjectvar.</td>
</tr>
<tr>
<td>index</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 adds 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.

Behavior change

In previous releases, calling the OpenTab function to open a user object as a tab page displayed the tab page even if the user object’s Visible property was set to false. In current releases, the user object’s Visible property must be set to true for the tab page to display.
Considerations when specifying a user object type

When you use Syntax 2, PowerBuilder opens an instance of a user object of the datatype specified in \texttt{userobjecttype} and places a reference to this instance in the variable \texttt{userobjectvar}. To refer to the instance in scripts, use \texttt{userobjectvar}.

If \texttt{userobjecttype} is a descendent user object, you can only refer to properties, events, functions, or structures that are part of the definition of \texttt{userobjectvar}. For example, if a user event is declared for \texttt{userobjecttype}, you cannot reference it.

The object specified in \texttt{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 opens a user object as the last tab page in the Tab control \texttt{tab_1}. The user object is of the type specified in the string \texttt{s_u_name} and stores the reference to the user object in the variable \texttt{u_to_open}:

```powershell
UserObject u_to_open
string s_u_name

s_u_name = sle_user.Text

\texttt{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 \texttt{OpenTab} does. \texttt{OpenTabWithParm} also stores a parameter in the system’s Message object so that it is accessible to the opened object.

<table>
<thead>
<tr>
<th>To open</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user object as a tab page</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>A user object as a tab page, allowing the application to select the user object’s type at runtime</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>
CHAPTER 10  PowerScript Functions

For user objects of a known datatype

Description
Opens a custom visual user object of a known datatype 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 datatype) or a variable of the desired user object datatype. OpenTabWithParm 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 datatypes:
• 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 datatype of the parameter specified for OpenTabWithParm, scripts for the opened user object would check one of the following properties.

Message object property  |  Argument datatype
--------------------------|--------------------------
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 immediately 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 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.

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 datatype
Opens a visual user object as a tab page within a Tab control when the datatype 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 )
```
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 datatype of the parameter specified for OpenTabWithParm, scripts for the opened user object would check one of the following properties.

<table>
<thead>
<tr>
<th>Message object property</th>
<th>Argument datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>message.DoubleParm</td>
<td>Numeric</td>
</tr>
<tr>
<td>message.PowerObjectParm</td>
<td>PowerObject (PowerBuilder objects, including user-defined structures)</td>
</tr>
<tr>
<td>message.StringParm</td>
<td>String</td>
</tr>
</tbody>
</table>

In the opened user object, it is a good idea to access the value passed in the Message object immediately 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.
OpenTabWithParm

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:

```powerbuilder
UserObject 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:

```powerbuilder
UserObject 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:

```powerbuilder
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

Add a user object to the specified window and makes all its properties and controls available to scripts.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open an instance of a particular user object</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Open a user object, allowing the application to select the user object’s type at runtime</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

For user objects of a known datatype

Description

Opens a user object of a known datatype.

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 datatype) or a variable of the desired user object datatype. OpenUserObject 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 be opened in a script. PowerBuilder opens it when it opens the window.
**OpenUserObject**

OpenUserObject adds 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 at runtime, 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 do not 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 in 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):

```powershell
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:

```powershell
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:

```powershell
w_info.OpenUserObject(u_data, 20, 100)
```

**See also**

OpenUserObjectWithParm
Syntax 2  
**For user objects of unknown datatype**

**Description**
Opens a user object when the datatype of the user object is not known until the script is executed.

**Applies to**
Window objects

**Syntax**
```
windowname.OpenUserObject ( userobjectvar, userobjecttype {, x, y} )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowname</td>
<td>The name of the window in which you want to open the user object.</td>
</tr>
<tr>
<td>userobjectvar</td>
<td>A variable of datatype DragObject. OpenUserObject places a reference to the opened user object in userobjectvar.</td>
</tr>
<tr>
<td>userobjecttype</td>
<td>A string whose value is the name of the user object you want to display. The datatype of userobjecttype must be a descendant of userobjectvar.</td>
</tr>
<tr>
<td>x (optional)</td>
<td>The x coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.</td>
</tr>
<tr>
<td>y (optional)</td>
<td>The y coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.</td>
</tr>
</tbody>
</table>

**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 be opened in a script. PowerBuilder opens it when it opens the window.

OpenUserObject adds 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 at runtime, 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 do not destroy the user object, it holds on to its allocated memory, resulting in a memory leak.
OpenUserObject

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 in 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 datatype 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.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open an instance of a particular user object</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Open a user object, allowing the application to select the user object’s type at runtime</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

**For user objects of a known datatype**

Description

Opens a user object of a known datatype and stores a parameter in the system’s Message object.

Applies to

Window objects

Syntax

```
windowname.OpenUserObjectWithParm ( userobjectvar, parameter {, x, y} )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowname</td>
<td>The name of the window in which you want to open the user object.</td>
</tr>
<tr>
<td>userobjectvar</td>
<td>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 datatype) or a variable of the desired user object datatype. OpenUserObjectWithParm places a reference to the opened user object in userobjectvar.</td>
</tr>
<tr>
<td>parameter</td>
<td>The parameter you want to store in the Message object when the user object is opened. Parameter must have one of these datatypes:</td>
</tr>
<tr>
<td></td>
<td>• String</td>
</tr>
<tr>
<td></td>
<td>• Numeric</td>
</tr>
<tr>
<td></td>
<td>• PowerObject</td>
</tr>
<tr>
<td>x (optional)</td>
<td>The x coordinate in PowerBuilder units of the user object within the window’s frame. The default is 0.</td>
</tr>
<tr>
<td>y (optional)</td>
<td>The y coordinate in PowerBuilder units of the user object within the window’s frame. The default is 0.</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, OpenUserObjectWithParm returns null.
OpenUserObjectWithParm

Usage

The system Message object has three properties for storing data. Depending on the datatype of the parameter specified for OpenUserObjectWithParm, scripts for the opened user object would check one of the following properties:

<table>
<thead>
<tr>
<th>Message object property</th>
<th>Argument datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>message.DoubleParm</td>
<td>Numeric</td>
</tr>
<tr>
<td>message.PowerObjectParm</td>
<td>PowerObject (PowerBuilder objects, including user-defined structures)</td>
</tr>
<tr>
<td>message.StringParm</td>
<td>String</td>
</tr>
</tbody>
</table>

In the opened user object, it is a good idea to access the value passed in the Message object immediately 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 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.

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, "Jim 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 datatype

Description
Opens a user object when the datatype 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
\( \text{windowname}.\text{OpenUserObjectWithParm} ( \text{userobjectvar}, \text{parameter}, \text{userobjecttype} \{, x, y \} ) \)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowname</td>
<td>The name of the window in which you want to open the user object.</td>
</tr>
<tr>
<td>userobjectvar</td>
<td>A variable of datatype DragObject. OpenUserObjectWithParm places a reference to the opened user object in userobjectvar.</td>
</tr>
<tr>
<td>parameter</td>
<td>The parameter you want to store in the Message object when the user object is opened. Parameter must have one of these datatypes:</td>
</tr>
<tr>
<td></td>
<td>• String</td>
</tr>
<tr>
<td></td>
<td>• Numeric</td>
</tr>
<tr>
<td></td>
<td>• PowerObject</td>
</tr>
<tr>
<td>userobjecttype</td>
<td>A string whose value is the datatype of the user object you want to open. The datatype of userobjecttype must be a descendant of userobjectvar.</td>
</tr>
<tr>
<td>x (optional)</td>
<td>The x coordinate in PowerBuilder units of the user object within the window’s frame. The default is 0.</td>
</tr>
<tr>
<td>y (optional)</td>
<td>The y coordinate in PowerBuilder units of the user object within the window’s frame. The default is 0.</td>
</tr>
</tbody>
</table>

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 datatype of the parameter specified for OpenUserObjectWithParm, scripts for the opened user object would check one of the following properties.

<table>
<thead>
<tr>
<th>Message object property</th>
<th>Argument datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>message.DoubleParm</td>
<td>Numeric</td>
</tr>
<tr>
<td>message.PowerObjectParm</td>
<td>PowerObject (PowerBuilder objects, including user-defined structures)</td>
</tr>
<tr>
<td>message.StringParm</td>
<td>String</td>
</tr>
</tbody>
</table>
In the opened user object, it is a good idea to access the value passed in the Message object immediately 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.

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`:

```powerbuilder
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`:

```powerbuilder
DragObject u_to_open
string s_u_name, e_location

e_location = sle_loc.Text
SELECT next_userobj INTO : s_u_name
FROM routing_table
WHERE ... ;

w_info.OpenUserObjectWithParm(u_to_open, &
  e_location, s_u_name)
```
The following statements display a user object of the type specified in the string $u\_name$ and store the reference to the user object in the variable $u\_to\_open$. The parameter is numeric so it is stored in message.DoubleParm. The user object is at the coordinates 100,200 in the window $w\_emp$:

```powerShell
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.

<table>
<thead>
<tr>
<th>To Use</th>
<th>Syntax 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open an instance of a particular window datatype</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Allow the application to select the window’s datatype when the script is executed</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1

**For windows of a known datatype**

Description

Opens a window object of a known datatype. 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

```powerShell
OpenWithParm( windowvar, parameter [, , parent ] )
```
**OpenWithParm**

**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 datatype) or a variable of the desired window datatype. OpenWithParm 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 datatypes:
- String
- Numeric
- PowerObject

`parent` | The window you want make the parent of the child or pop-up window you are opening. If you open a child or pop-up window and omit `parent`, PowerBuilder associates the window being opened with the currently active window.

( child and pop-up windows only) (optional)

**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 datatype of the parameter specified for OpenWithParm, your scripts for the opened window would check one of the following properties.

<table>
<thead>
<tr>
<th>Message object property</th>
<th>Argument datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message.DoubleParm</td>
<td>Numeric</td>
</tr>
<tr>
<td>Message.PowerObjectParm</td>
<td>PowerObject (PowerBuilder objects, including user-defined structures)</td>
</tr>
<tr>
<td>Message.StringParm</td>
<td>String</td>
</tr>
</tbody>
</table>

In the opened window, it is a good idea to access the value passed in the Message object immediately 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.

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_employee. 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
OpenWithParm

**Syntax 2**

**For windows of unknown datatype**

Opens a window object when you do not know its datatype 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 } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windowvar</code></td>
<td>A window variable, usually of datatype window. OpenWithParm places a reference to the open window in <code>windowvar</code>.</td>
</tr>
<tr>
<td><code>parameter</code></td>
<td>The parameter you want to store in the Message object when the window is opened. Parameter must have one of these datatypes:</td>
</tr>
<tr>
<td></td>
<td>• String</td>
</tr>
<tr>
<td></td>
<td>• Numeric</td>
</tr>
<tr>
<td></td>
<td>• PowerObject</td>
</tr>
<tr>
<td><code>windowtype</code></td>
<td>A string whose value is the datatype of the window you want to open. The datatype of <code>windowtype</code> must be the same or a descendant of <code>windowvar</code>.</td>
</tr>
<tr>
<td><code>parent</code></td>
<td>(child and pop-up windows only) The window you want to make the parent of the child or pop-up window you are opening. If you open a child or pop-up window and omit <code>parent</code>, PowerBuilder associates the window being opened with the currently active window.</td>
</tr>
</tbody>
</table>

**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 datatype of the parameter specified for OpenWithParm, your scripts for the opened window would check one of the following properties.

<table>
<thead>
<tr>
<th>Message object property</th>
<th>Argument datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message.DoubleParm</td>
<td>Numeric</td>
</tr>
<tr>
<td>Message.PowerObjectParm</td>
<td>PowerObject (PowerBuilder objects, including user-defined structures)</td>
</tr>
<tr>
<td>Message.StringParm</td>
<td>String</td>
</tr>
</tbody>
</table>
In the opened window, it is a good idea to access the value passed in the Message object immediately 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.

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 \texttt{s\_w\_name} and store the reference to the window in the variable \texttt{w\_to\_open}. The script gets the value of \texttt{s\_w\_name}, the type of window to open, from the database. The parameter in \texttt{e\_location} is text, so it is stored in Message.StringParm:

```powerobject
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 \texttt{c\_w\_name}, store the reference to the window in the variable \texttt{wc\_to\_open}, and make \texttt{w\_emp} the parent window of \texttt{wc\_to\_open}. The parameter is numeric, so it is stored in \texttt{Message.DoubleParm}:

\begin{verbatim}
window wc_to_open
string c_w_name
integer age = 60

  c_w_name = "w\_c\_empl"

  OpenWithParm(wc_to_open, age, c_w_name, w_emp)
\end{verbatim}

See also

CloseWithReturn
Open

\section*{OutgoingCallList}

\textbf{Description}\hfill Provides a list of the calls to other routines included in a performance analysis model.

\textbf{Applies to}\hfill ProfileLine and ProfileRoutine objects

\textbf{Syntax}\hfill \textit{instancename.\textbf{OutgoingCallList} ( list, aggregate )}

\begin{tabular}{|l|l|}
\hline
\textbf{Argument} & \textbf{Description} \\
\hline
\textit{instancename} & Instance name of the ProfileLine or ProfileRoutine object. \\
\hline
\textit{list} & An unbounded array variable of datatype ProfileCall in which \textbf{OutgoingCallList} stores a ProfileCall object for each call to other routines from within this routine. This argument is passed by reference. \\
\hline
\textit{aggregate} & A boolean indicating whether duplicate routine calls will result in the creation of a single or of multiple ProfileCall objects. \texttt{(ProfileRoutine only)} \\
\hline
\end{tabular}

\textbf{Return value}\hfill 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 aggregate argument indicates whether duplicate routine calls 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 aggregate is true, a new ProfileCall object is created that aggregates all calls from the calling routine to the current routine. If aggregate 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:

```powerscript
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
rtnname.PageCount ( )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtnname</td>
<td>The name of the RichTextEdit control in which you want the page count</td>
</tr>
</tbody>
</table>

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 method for DataWindows in the DataWindow Reference or the online Help

PageCreated

Description
Reports whether a tab page has been created.

Applies to
User objects used as tab pages

Syntax
userobject.PageCreated ( )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userobject</td>
<td>The name of the tab page whose existence you want to test</td>
</tr>
</tbody>
</table>

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 has not 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:

```powerbuilder
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**

```powerbuilder
windowname.ParentWindow()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windowname</code></td>
<td>The name of a window for which you want to obtain the parent object</td>
</tr>
</tbody>
</table>

**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—the local environment. `ParentWindow` returns the parent window of a specified 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 do not 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.
Pop-up windows have a parent window when they are opened from another window but when used in an MDI application, the parent of the pop-up is the MDI frame. A pop-up 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 with which the menu is associated when your program is running. For more information, see the PowerBuilder Users Guide.

Examples

These statements return the parent of child_1. The parent is a window of the datatype Win1:

```powerbuilder
Win1 w_parent
w_parent = child_1.ParentWindow()
```

The following script for a Cancel button in a pop-up 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:

```powerbuilder
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.

**Applies to**

EditMask, InkEdit, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, DropDownListPictureListBox, DataWindow, OLE controls

**Syntax**

```powerbuilder
controlnamePaste ()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the DataWindow control, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, DropDownListPictureListBox, or OLE control into which you want to insert the contents of the clipboard. If <code>controlname</code> is a DataWindow, text is pasted into the edit control over the current row and column. If <code>controlname</code> is a DropDownListBox or DropDownListPictureListBox, the AllowEdit property must be true.</td>
</tr>
</tbody>
</table>
Return value

Integer for DataWindow, InkEdit, and list boxes, Long for other controls.

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 content 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 does not match the column’s datatype, 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:

do_1.Paste()
PasteLink

Description  Pastes a link to the contents of the clipboard into the control. The server application for the object on the clipboard must be running.

Applies to  OLE controls

Syntax  

\[
\text{olecontrol}.\text{PasteLink}() 
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olecontrol</td>
<td>The name of the OLE control into which you want to paste the object on the clipboard</td>
</tr>
</tbody>
</table>

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 \text{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 do not 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 runs the server application in the background to enable the link.

PasteLink fails, however, 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.

See also  Copy
Cut
PasteLink
PasteSpecial
ReplaceText
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 `li_result` to 0:

```power_script
integer li_result
li_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**

```power_script
rtname.PasteRTF ( richtextstring, { band } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtname</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>richtextstring</code></td>
<td>A string whose value is data with rich text formatting.</td>
</tr>
<tr>
<td><code>band</code> (optional)</td>
<td>A value of the Band enumerated datatype specifying the band into which the rich text data is pasted. Values are:</td>
</tr>
<tr>
<td></td>
<td>• Detail! – The data is pasted into the detail band</td>
</tr>
<tr>
<td></td>
<td>• Header! – The data is pasted into the header band</td>
</tr>
<tr>
<td></td>
<td>• Footer! – The data is pasted into the footer band</td>
</tr>
<tr>
<td></td>
<td>The default is the band that contains the insertion point.</td>
</tr>
</tbody>
</table>

**Return value**

Long. Returns -1 if an error occurs. If `richtextstring` is null, `PasteRTF` returns null.

**Usage**

A DataWindow in the RichText 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`:

```power_script
string ls_richtext
rte_message.PasteRTF(ls_richtext, Header!)
```
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.

Applies to OLE controls

Syntax `olecontrol.PasteSpecial()`

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>User canceled without selecting a paste option</td>
</tr>
<tr>
<td>-1</td>
<td>No data found</td>
</tr>
<tr>
<td>-9</td>
<td>Other error</td>
</tr>
</tbody>
</table>

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`:

```powerbuilder
integer li_result
li_result = ole_1.PasteSpecial()
```

See also LinkTo
Paste
PasteLink

See also CopyRTF
PBGetMenuString

Description
Gets the name of the item at a given position in a menu.

Syntax
PBGetMenuString (hmenu, nPos, caption, nMaxLength)

Argument | Description
--- | ---
hmenu | A long for the menu handle
nPos | An integer for the position of the menu item, counting from 0 at the leftmost or topmost position
caption | A string passed by reference that captures the name (Text property) of the menu item
nMaxLength | An integer that sets the maximum length of the value passed in the caption argument

Return value
Long. Returns 1 if it succeeds and -1 if an error occurs. If any argument is null, PBGetMenuString returns null.

Usage
Use PBGetMenuString to get the name of a menu item. This function is useful for some automated testing programs that cannot get menu item names from the Text property for menus that use the contemporary style.

Examples
This statement gets the first menu item of the submenu of the w_main window using the GetMenu and GetSubMenu WIN32 API functions from the user.dll library:

```powerShell
string ls_menu
long hmenu
long submenu
int ll_ret

hmenu = GetMenu(handle(w_main))
submenu = GetSubMenu(hmenu,0)
ll_ret = PBGetMenuString(submenu,0,ls_menu,5)
messagebox ("Menu Test", "return value = &" +string(ll_ret)+ " menu caption is "+ls_menu)
```

This example assumes you have made the following Local External Functions declarations:

```powerShell
function long GetMenu ( long hwnd) library "user32.dll"
function long GetSubMenu ( long hparent,int pos) &
    library "user32.dll"
```
**Pi**

**Description**
Multiplies pi by a specified number.

**Syntax**

```
Pi ( n )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The number you want to multiply by pi (3.14159265358979323...)</td>
</tr>
</tbody>
</table>

**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 method for DataWindows in the DataWindow Reference or the online Help

---

**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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pixels</td>
<td>An integer whose value is the number of pixels you want to convert to PowerBuilder units.</td>
</tr>
</tbody>
</table>

---

824  
PowerBuilder
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:

```powerbuilder
integer Value
Value = PixelsToUnits(35, XPixelsToUnits!)
```

See also UnitsToPixels

---

## Play

**Description** Starts playing an animation (an AVI clip).

**Applies to** Animation controls

**Syntax**

```powerbuilder
animationname.Play ( from, to, replay )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animationname</td>
<td>The name of the animation control displaying the AVI clip.</td>
</tr>
<tr>
<td>from</td>
<td>A long value in the range 0 to 65,535 indicating the frame where playing starts. The value 0 starts playing the clip at the first frame.</td>
</tr>
<tr>
<td>to</td>
<td>A long value in the range -1 to 65,535 indicating the frame where playing ends. The value -1 stops playing the clip at the last frame.</td>
</tr>
<tr>
<td>replay</td>
<td>A long value in the range -1 to 65,535 indicating the number of times to replay the clip. The value -1 continues playing the clip indefinitely.</td>
</tr>
</tbody>
</table>

**Return value** Integer. Returns 1 for success and -1 for failure.

**Usage** Start plays an opened AVI file in an animation control. If you specify a value for any argument that is not in the specified range, Start does nothing and returns -1.
### PointerX

#### Description
Determine 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 do not specify `objectname`, `PointerX` reports the distance from the left edge of the current sheet or window.

#### Return value
Integer. Returns the pointer’s distance from the left edge of `objectname` in PowerBuilder units if it succeeds and -1 if an error occurs.

#### Examples
In a script for a control in a window, the following example stores the distance of the pointer from the edge of the window in the variable `li_dist`. If the pointer is 5 units from the left edge of the window, `li_dist` equals 5:

```
integer li_dist
li_dist = Parent.PointerX()
```

This statement in a control’s `RButtonDown` script displays a pop-up menu 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 pop-up menu at the cursor position:

```
m_Appl.m_Help.PopMenu(This.PointerX(), & This.PointerY())
```

#### See also
`PointerY`
CHAPTER 10  PowerScript Functions

PowerScript Reference

827

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)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectname</code></td>
<td>The name of the control or window for which you want the pointer's distance from the top. If you do not specify <code>objectname</code>, <code>PointerY</code> reports the distance from the top of the current sheet or window.</td>
</tr>
</tbody>
</table>

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:

```powerbuilder
integer li_Dist
li_Dist = Parent.PointerY()
```

This statement in a control's RButtonDown script displays a pop-up menu at the cursor position:

```powerbuilder
```

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)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuname</td>
<td>The fully qualified name of a menu on a menu bar you want to display at the specified location</td>
</tr>
<tr>
<td>xlocation</td>
<td>The distance in PowerBuilder units of the displayed menu from the left edge of the window</td>
</tr>
<tr>
<td>ylocation</td>
<td>The distance in PowerBuilder units of the displayed menu from the top of the window</td>
</tr>
</tbody>
</table>

**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 pop-up 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 pop-up 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 (datatype 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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>number</code></td>
<td>The integer to be stored in the number property of the Error object</td>
</tr>
<tr>
<td><code>text</code></td>
<td>The string to be stored in text property of the Error object</td>
</tr>
</tbody>
</table>

**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 have created. If there is no script for the SystemError event, the SignalError function does nothing.

**Examples**

The `gf_DoSomething` function takes a table name and a record and returns 0 for success and a negative number for an error. 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:

```
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 } )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td>The string in which you want to find string2.</td>
</tr>
<tr>
<td>string2</td>
<td>The string you want to find in string1.</td>
</tr>
<tr>
<td>start</td>
<td>A long indicating where the search will begin in string1. The default is 1.</td>
</tr>
</tbody>
</table>

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, because the case does not match:

```
Pos("BABE RUTH", "be")
```

This statement starts searching at position 4 and returns 0, because position 4 is after the occurrence of BE:

```
Pos("BABE RUTH", "BE", 4)
```

These statements change the text NY in the SingleLineEdit sle_group to North East:

```powerShell
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:

```powerShell
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 method for DataWindows in the DataWindow Reference or the online Help
- GetObjectAtPointer method for DataWindows in the DataWindow Reference or the online Help
- LastPos
- Left
- Mid
- Right
- Pos method for DataWindows in the DataWindow Reference or the online Help
**PosA**

**Description**
Temporarily converts a string from Unicode to DBCS based on the current locale, then finds one string within another string.

**Syntax**

```
PosA (string1, string2, {start})
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td>The string in which you want to find string2.</td>
</tr>
<tr>
<td>string2</td>
<td>The string you want to find in string1.</td>
</tr>
<tr>
<td>start (optional)</td>
<td>A long indicating the position in string1 where the search will begin. The position is indicated by the number of bytes you specify for this argument. The default is 1.</td>
</tr>
</tbody>
</table>

**Return value**
Long. Returns a long whose value is the starting position of the first occurrence of string2 in string1 after the position in bytes specified by start. If string2 is not found in string1 or if start is not within string1, PosA returns 0. If any argument’s value is null, PosA returns null.

**Usage**
PosA replaces the functionality that Pos had in DBCS environments in PowerBuilder 9. In SBCS environments, Pos, PosW, and PosA return the same results.

**PosW**

**Description**
Finds one string within another string. This function is obsolete. It has the same behavior as Pos in all environments.

**Syntax**

```
PosW (string1, string2 {, start})
```

**Position**

Reports the position of the insertion point in an editable control.

<table>
<thead>
<tr>
<th>To report</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>The position of the insertion point in any editable control (except RichTextEdit)</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>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</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>
**Syntax 1**

For **editable controls, except RichTextEdit**

**Description**
Determines the position of the insertion point in an edit control.

**Applies to**
DataWindow, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, or DropDownListBox, DropDownPictureListBox controls

**Syntax**

```
editname.Position ()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>editname</code></td>
<td>The name of the DataWindow control, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, or DropDownListBox, DropDownPictureListBox control in which you want to find the location of the insertion point</td>
</tr>
</tbody>
</table>

**Return value**
Integer for DataWindow, InkEdit, and list boxes, Long for other controls.

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
Position

### 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 a RichTextEdit control.

**Applies to**

RichTextEdit and DataWindow controls

**Syntax**

```
rtename.Position ( fromline, fromchar [, toline, tochar ] )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>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.</td>
</tr>
<tr>
<td>fromline</td>
<td>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.</td>
</tr>
<tr>
<td>fromchar</td>
<td>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.</td>
</tr>
<tr>
<td>toline</td>
<td>(optional) A long variable in which you want to save the number of the line where the selection ends.</td>
</tr>
<tr>
<td>tochar</td>
<td>(optional) A long variable in which you want to save the number in the line of the character before which the selection ends.</td>
</tr>
</tbody>
</table>

**Return value**

Band enumerated datatype. 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 datatype 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
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.

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
Add 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 } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectname</code></td>
<td>The name of any PowerBuilder object or control (except an application) that has events associated with it.</td>
</tr>
<tr>
<td><code>event</code></td>
<td>A value of the TrigEvent enumerated datatype 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 <code>objectname</code> and a script must exist for the event in <code>objectname</code>.</td>
</tr>
<tr>
<td><code>word</code></td>
<td>(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 <code>long</code>, but not <code>word</code>, enter 0. (For cross-platform compatibility, WordParm and LongParm are both longs).</td>
</tr>
<tr>
<td><code>long</code></td>
<td>(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).</td>
</tr>
</tbody>
</table>

Return value
Boolean. Returns true if it is successful and false if the event is not a valid event for `objectname`. Also returns true if 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 do not 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:

```powershell
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). PowerBuilder maps the message to its internal close message (PBM_CLOSE), then 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 did not 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, 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, {serverport, } data )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>servicereference</code></td>
<td>Reference to the Internet service instance.</td>
</tr>
<tr>
<td><code>urlname</code></td>
<td>String specifying the URL to post.</td>
</tr>
</tbody>
</table>

---

PowerScript Reference 841
PostURL

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>urldata</td>
<td>Blob specifying arguments to the URL specified by urlname.</td>
</tr>
<tr>
<td>headers</td>
<td>String specifying HTML headers. In Netscape, a newline (~n) is required after each HTTP header and a final newline after all headers.</td>
</tr>
<tr>
<td>serverport (optional)</td>
<td>Specifies the server port number for the request. The default value for this argument is 0, which means that the port number is determined by the system (port 80 for HTTP requests).</td>
</tr>
<tr>
<td>data</td>
<td>InternetResult instance into which the function returns HTML.</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns values as follows:

- 1 Success
- -1 General error
- -2 Invalid URL
- -4 Cannot connect to the Internet
- -5 Unsupported secure (HTTPS) connection attempted
- -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 required processing 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).

To simulate a form submission, you need to send a header that indicates the proper Content-Type. For forms, the proper Content-Type header is:

```
Content-Type: application/x-www-form-urlencoded
```

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 using server port 8080. Iinet is an instance variable of type inet:

```powershell
Blob lbbl_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?"
```
CHAPTER 10  PowerScript Functions

```
lblb_args = blob("\n")
ll_length = Len(lblb_args)
ls_headers = "Content-Length: " &
    + String(ll_length) + "-n-n"
inet.PostURL &
    (ls_url, lblb_args, ls_headers, 8080, iir_msgbox)
```

This example shows the use of a header with the correct content-type for a form:

```
Blob lblb_args
String ls_headers
String ls_url
String ls_args
long ll_length
integer li_rc

li_rc = GetContextService( "Internet", iinet_base )
IF li_rc = 1 THEN
    ir = CREATE n_ir
    ls_url = "http://localhost/Site/testurl.stm?"
    ls_args = "user=MyName&pwd=MyPasswd"
    lblb_args = Blob( ls_args )
    ll_length = Len( lblb_args )
    ls_header = "Content-Type: " + &
        "application/x-www-form-urlencoded-n" + &
        "Content-Length: " + String( ll_length ) + "-n-n"
    li_rc = iinet.PostURL( ls_url, lblb_args, &
        ls_header, ir )
END IF
```

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 )
```
**Preview**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control which you want to preview or edit.</td>
</tr>
<tr>
<td>previewsetting</td>
<td>A boolean value indicating whether to put the RichTextEdit into preview or edit mode. Values are:</td>
</tr>
<tr>
<td></td>
<td>• True – Preview the contents of the RichTextEdit as it would look when printed</td>
</tr>
<tr>
<td></td>
<td>• False – Displays the contents in editable form</td>
</tr>
</tbody>
</table>

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.

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)
```

See also

`IsPreview`
Print

Sends data to the current printer (or spooler, if the user has a spooler set up). There are several syntaxes.

For syntax for DataWindows or DataStores, see the Print method for DataWindows in the *DataWindow Reference* or the online Help.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include a visual object, such as a window or a graph control in a print job</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Send one or more lines of text as part of a print job</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Print the contents of an RTE control</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

Syntax 1  
**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.
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.

PowerBuilder copies the area of the screen occupied by the control to the printer. If any other window or application displays on the screen in that area in front of the control while the control is being printed, that window or application will also be printed.

**Print area and margins** The print area is the physical page size minus any margins in the printer itself.

### Examples

This example prints the CommandButton cb_close in its original size at location 500, 1000:

```powershell
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:

```powershell
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)
```

### Syntax 2

**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
### Print

Syntax: `Print ( printjobnumber, { tab1, } string {, tab2 } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printjobnumber</code></td>
<td>The number the PrintOpen function assigned to the print job.</td>
</tr>
<tr>
<td><code>tab1</code> (optional)</td>
<td>The position, measured from the left edge of the print area in thousandths of a inch, to which the print cursor should move before <code>string</code> is printed. If the print cursor is already at or beyond the position or if you omit <code>tab1</code>, Print starts printing at the current position of the print cursor.</td>
</tr>
<tr>
<td><code>string</code></td>
<td>The string you want to print. If the string includes carriage return-newline character pairs (<del>r</del>n), the string will print on multiple lines. However, the initial tab position is ignored on subsequent lines.</td>
</tr>
<tr>
<td><code>tab2</code> (optional)</td>
<td>The new position, measured from the left edge of the print area in thousandths of a inch, of the print cursor after <code>string</code> printed. If the print cursor is already at or beyond the specified position, Print ignores <code>tab2</code> and the print cursor remains at the end of the text. If you omit <code>tab2</code>, Print moves the print cursor to the beginning of a new line.</td>
</tr>
</tbody>
</table>

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 do not 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

Print area and margins  The print area is the physical page size minus any margins in the printer itself.

Using fonts  You can use PrintDefineFont and PrintSetFont to specify the font used by the Print function when you are printing a string.

Fonts for multiple languages  The default font for print functions is the system font, but multiple languages cannot be printed correctly using the system font. The Tahoma font typically produces good results. However, if the printer font is set to Tahoma and the Tahoma font is not installed on the printer, PowerBuilder downloads the entire font set to the printer when it encounters a multilanguage character. Use the PrintDefineFont and PrintSetFont functions to specify a font that is available on users’ printers and supports multiple languages.

Examples

This example opens a print job, prints the string Sybase Corporation in the default font, and then starts a new line:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
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)
Print

See also  PrintCancel  
PrintClose  
PrintDataWindow  
PrintOpen  
PrintScreen  
PrintSetFont  
PrintSetSpacing

Syntax 3  For RichTextEdit controls
Description  Prints the contents of a RichTextEdit control.
Applies to  RichTextEdit controls
Syntax  \textit{rtename}.Print \textit{( copies, pagerange, collate, canceldialog )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{rtename}</td>
<td>The name of the RichTextEdit control whose contents you want to print.</td>
</tr>
<tr>
<td>\textit{copies}</td>
<td>An integer specifying the number of copies you want to print.</td>
</tr>
<tr>
<td>\textit{pagerange}</td>
<td>A string describing the pages you want to print. To print all pages, specify an empty string (&quot;&quot;). To specify a subset of pages, use dashes to specify a range and commas to separate ranges and individual page numbers—for example, &quot;1-3&quot; or &quot;2,5,8-10&quot;. When \textit{rtename} shares data with a DataWindow, \textit{pagerange} refers to pages based on the total number of pages in the control, not within each instance of the document.</td>
</tr>
<tr>
<td>\textit{collate}</td>
<td>A boolean value indicating whether you want the copies collated. Values are:</td>
</tr>
<tr>
<td></td>
<td>\texttt{TRUE} – Collate copies</td>
</tr>
<tr>
<td></td>
<td>\texttt{FALSE} – Do not collate copies</td>
</tr>
<tr>
<td>\textit{canceldialog}</td>
<td>A boolean value indicating whether you want to display a nonmodal dialog box that allows the user to cancel printing. Values are:</td>
</tr>
<tr>
<td></td>
<td>\texttt{TRUE} – Display the dialog box</td>
</tr>
<tr>
<td></td>
<td>\texttt{FALSE} – Do not display the dialog box</td>
</tr>
</tbody>
</table>

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.
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 box displays to allow the user to cancel the printing:

\[
\text{rte}_1.\text{Print}(1, "1-5", \text{FALSE}, \text{TRUE})
\]

See also

Preview
PrintEx

PrintBitmap

Description

Writes a bitmap at the specified location on the current page.

Syntax

\[
\text{PrintBitmap} \left( \text{printjobnumber, bitmap, x, y, width, height} \right)
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printjobnumber</td>
<td>The number the PrintOpen function assigned to the print job.</td>
</tr>
<tr>
<td>bitmap</td>
<td>A string whose value is the file name of the bitmap image.</td>
</tr>
<tr>
<td>x</td>
<td>An integer whose value is the x coordinate (in thousandths of an inch) on the page of the bitmap image.</td>
</tr>
<tr>
<td>y</td>
<td>An integer whose value is the y coordinate (in thousandths of an inch) on the page of the bitmap image.</td>
</tr>
<tr>
<td>width</td>
<td>The integer width of the bitmap image in thousandths of an inch. If width is 0, PowerBuilder uses the original width of the image.</td>
</tr>
<tr>
<td>height</td>
<td>The integer height of the bitmap image in thousandths of an inch. If height is 0, PowerBuilder uses the original height of the image.</td>
</tr>
</tbody>
</table>

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 \texttt{d:\PB\BITMAP1.BMP} in its original size at location 50,100:

\[
\text{long Job}
\]

\[
// \text{Define a new blank page.}
\text{Job} = \text{PrintOpen}();
\]
// Print the bitmap in its original size.
PrintBitmap(Job, "d:\PB\BITMAP1.BMP", 50,100, 0,0)
// Send the page to the printer and close Job.
PrintClose(Job)

See also
PrintClose
PrintLine
PrintRect
PrintRoundRect
PrintOval
PrintOpen

PrintCancel

Description
Cancels printing and deletes the spool file, if any. Cancels printing of a print
job that you opened with the PrintOpen function. The print job is identified by
the number returned by PrintOpen.

For syntax for DataWindows and DataStores, see the PrintCancel method for
DataWindows in the DataWindow Reference or the online Help.

Syntax
PrintCancel ( printjobnumber )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printjobnumber</td>
<td>The number the PrintOpen function assigned to the print job</td>
</tr>
</tbody>
</table>

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:

```plaintext
boolean gb_printcancel
```

The script for the Print button is:

```plaintext
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:

```plaintext
gb_printcancel = TRUE
Close(w_printcancel)
```

See also

- Print
- PrintClose
- PrintOpen
PrintClose

Description
Sends the current page to the printer (or spooler) and closes the job. Call PrintClose as the last command of a print job unless PrintCancel function has closed the job.

Syntax
PrintClose ( printjobnumber )

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)
```

See also
PrintCancel
PrintOpen
PrintDataWindow

Description
Prints the contents of a DataWindow control or DataStore as a print job.

Syntax
PrintDataWindow( printjobnumber, dwcontrol )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printjobnumber</td>
<td>The number the PrintOpen function assigned to the print job</td>
</tr>
<tr>
<td>dwcontrol</td>
<td>The name of the DataWindow control, child DataWindow, or DataStore containing the DataWindow object you want to print</td>
</tr>
</tbody>
</table>

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 using PrintDataWindow, PowerBuilder uses the fonts and layout specified in the computer’s printer setup, not the fonts and layout specified in the DataWindow. The PrintDefineFont and PrintSetFont methods also have no effect.

When the DataWindow’s presentation style is RichTextEdit, each row begins a new page in the printed output.

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:

```powerscript
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printjobnumber</td>
<td>The number the PrintOpen function assigned to the print job.</td>
</tr>
<tr>
<td>fontnumber</td>
<td>The number (1 to 8) you want to assign to the font.</td>
</tr>
<tr>
<td>facename</td>
<td>A string whose value is the name of a typeface supported by your printer (for example, Courier 10Cpi).</td>
</tr>
<tr>
<td>height</td>
<td>An integer whose value is 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.</td>
</tr>
<tr>
<td>weight</td>
<td>The stroke weight of the type. Normal weight is 400 and bold is 700.</td>
</tr>
<tr>
<td>fontpitch</td>
<td>A value of the FontPitch enumerated datatype indicating the pitch of the font: Default! Fixed! Variable!</td>
</tr>
<tr>
<td>fontFamily</td>
<td>A value of the FontFamily enumerated datatype indicating the family of the font: AnyFont! Decorative! Modern! Roman! Script! Swiss!</td>
</tr>
<tr>
<td>italic</td>
<td>A boolean value indicating whether the font is italic. The default is false (not italic).</td>
</tr>
<tr>
<td>underline</td>
<td>A boolean value indicating whether the font is underlined. The default is false (not underlined).</td>
</tr>
</tbody>
</table>

**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.

**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 does not 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:

```power Cruise```
```printdefinefont```
```pntcruise```
```height```
```decorative```
```false```
```false```

### See also
- PrintClose
- PrintOpen
- PrintSetFont

---

**PrintEx**

**Description**
Prints the contents of a RichTextEdit control.

**Applies to**
RichTextEdit controls

**Syntax**

```
rtename.PrintEx ( canceldialog )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</code></td>
<td>The name of the RichTextEdit control whose contents you want to print.</td>
</tr>
<tr>
<td><code>canceldialog</code></td>
<td>A boolean value indicating whether you want to display a nonmodal Cancel dialog box that allows the user to cancel printing. The System Print dialog box always displays. Values are:</td>
</tr>
<tr>
<td></td>
<td>TRUE – Display the dialog box</td>
</tr>
<tr>
<td></td>
<td>FALSE – Do not display the dialog box</td>
</tr>
</tbody>
</table>
PrintGetPrinter

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If the user presses Cancel in the Print dialog box, PrintEx returns -1. If the user presses Cancel in the Cancel dialog box, PrintEx returns 1.

Usage

To specify a range of pages and the number of copies to print and whether pages should be collated, use the Print function.

Examples

This statement prints the document in the RichTextEdit control rte_1. A Cancel dialog box displays to allow the user to cancel the printing:

```
rte_1.PrintEx(TRUE)
```

See also

Preview
Print

PrintGetPrinter

Description

Gets the current printer name.

Syntax

`PrintGetPrinter()`

Return value

String. Returns current printer information in a tab-delimited format:
`printername ~t drivername ~t port`.

Usage

The current printer is the default printer unless you change it with the PrintSetPrinter method. A PowerBuilder application calling the PrintGetPrinter method does not get an externally reset default after the application initializes.

Examples

This example places the current printer name, driver, and port in separate SingleLineEdit textboxes:

```
String ls_fullstring=PrintGetPrinter()
String ls_name, ls_driver, ls_port, ls_temp
Long ll_place

ll_place=pos (ls_fullstring, "~t")
ls_name=left(ls_fullstring, ll_place -1)
ls_temp=mid(ls_fullstring, ll_place +1)
ll_place=pos (ls_temp, "-t")
ls_driver=left(ls_temp, ll_place -1)
ls_port=mid(ls_temp, ll_place +1)
sle_1.text=ls_name
sle_2.text=ls_driver
sle_3.text=ls_port
```

See also

PrintGetPrinters
PrintSetPrinter
PrintGetPrinters

Description
Gets the list of available printers.

Syntax
PrintGetPrinters()

Return value
String. Each printer is listed in the string in the format
printername ~t
drivename ~t port ~n.

Usage
The return string can be loaded into a DataWindow using ImportString or
separated using the ~n as shown in the example.

Examples
This example parses printer names from the return string on the
PrintGetPrinters call, then places each printer name in an existing
SingleLineEdit control. If you have more printers than SingleLineEdit boxes,
the last SingleLineEdit contains a string for all the printers that are not listed in
the other SingleLineEdits:

```powerscript
singlelineedit sle
long ll_place, i, k
string ls_left, ls_prntrs

ls_prntrs = PrintGetPrinters()
k = upperbound(control)
FOR i = k to 1 STEP -1
   IF parent.control[i].typeof() = singlelineedit then
      sle = parent.control[i]
      ll_place = pos(ls_prntrs,~-n~)
      ls_left = Left(ls_prntrs, ll_place - 1)
      sle.text = ls_left
      ls_prntrs = Mid(ls_prntrs, ll_place + 1)
   END IF
NEXT
sle.text = ls_prntrs
```

See also
ImportString method for DataWindows in the DataWindow Reference or the
online Help
PrintGetPrinter
PrintSetPrinter
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.

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:

```powerbuilder
long Job = PrintOpen( )
... // Various print commands
// Start a new page.
PrintPage(Job)
// Print a line at the top of the page
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, showprintdialog } )

Argument | Description
--- | ---
jobname (optional) | A string specifying a name for the print job. The name is displayed in the Windows Print Manager dialog box and in the Spooler dialog box.
showprintdialog (optional) | A boolean value indicating whether you want to display the system Print dialog box that allows the user to select a printer or set print properties. Values are:
TRUE – Display the dialog box
FALSE – (default) Do not display the dialog box

Return value
Long. Returns the job number if it succeeds and -1 if an error occurs. If the Print dialog box displays and the user presses Cancel, PrintOpen returns -1. 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.

If you specify true for the showprintdialog argument, the system Print dialog box displays allowing the user to cancel the print job. The option to specify a page range in the Print dialog box is disabled because PowerBuilder cannot determine the number of pages in the print job in advance. If you specify this argument in a component that runs on a server, the argument is ignored.

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.
This example opens a job but does not give it a name:

```c
ulong li_job
li_job = PrintOpen()
```

This example opens a job, gives it a name, and displays the Print dialog box:

```c
ulong li_job
li_job = PrintOpen("Phone List", true)
```

**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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printjobnumber</code></td>
<td>The number the PrintOpen function assigned to the print job</td>
</tr>
<tr>
<td><code>x</code></td>
<td>An integer specifying the x coordinate in thousandths of an inch of the upper-left corner of the oval’s bounding box</td>
</tr>
<tr>
<td><code>y</code></td>
<td>An integer specifying the y coordinate in thousandths of an inch of the upper-left corner of the oval’s bounding box</td>
</tr>
</tbody>
</table>

---

862  PowerBuilder
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.

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:

```plaintext
long Job

// Define a new blank page.
Job = PrintOpen()

// Print an oval.
PrintOval(Job, 4000, 3000, 1000, 1000, 10)

... // Other printing
PrintClose(Job)
```
**PrintPage**

**Description**
Sends the current page to the printer or spooler and begins a new blank page in the current print job.

**Syntax**

```plaintext
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:

```plaintext
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")
```

**See also**
PrintClose
PrintOpen

---

**PrintPage**

**Description**
Sends the current page to the printer or spooler and begins a new blank page in the current print job.

**Syntax**

```plaintext
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:

```plaintext
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")
```

**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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printjobnumber</td>
<td>The number the PrintOpen function assigned to the print job</td>
</tr>
<tr>
<td>x</td>
<td>An integer specifying the x coordinate in thousandths of an inch of the upper-left corner of the rectangle</td>
</tr>
<tr>
<td>y</td>
<td>An integer specifying the y coordinate in thousandths of an inch of the upper-left corner of the rectangle</td>
</tr>
<tr>
<td>width</td>
<td>An integer specifying the rectangle’s width in thousandths of an inch</td>
</tr>
<tr>
<td>height</td>
<td>An integer specifying the rectangle’s height in thousandths of an inch</td>
</tr>
<tr>
<td>thickness</td>
<td>An integer specifying the thickness of the rectangle’s border line in thousandths of an inch</td>
</tr>
</tbody>
</table>

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.

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:

```powerScript
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)
```
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.

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:

```powerScript
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

```powerScript
PrintScreen ( printjobnumber, x, y {, width, height } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printjobnumber</td>
<td>The number the PrintOpen function assigns to the print job.</td>
</tr>
<tr>
<td>x</td>
<td>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.</td>
</tr>
</tbody>
</table>
PrintSend

Argument | Description
--- | ---
y | An integer whose value is the y coordinate on the page, in thousandths of an inch, of the upper-left corner of the screen image.
width (optional) | The integer width of the printed screen in thousandths of an inch. If you omit width, PowerBuilder prints the screen at its original width. 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:

```powershell
long 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.

Obsolete function
PrintSend is an obsolete function and is provided for backward compatibility only. The ability to use this function is dependent upon the printer driver.

Syntax

```powershell
PrintSend ( printjobnumber, string, (zerochar) )
```
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 do not 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:

```powerbuilder
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.
*/
```
PrintSetFont

**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 ( printkonenumber, fontnumber )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printkonenumber</td>
<td>The number the PrintOpen function assigned to the print job</td>
</tr>
<tr>
<td>fontnumber</td>
<td>The number (1 to 8) of a font defined for the job in PrintDefineFont or 0 (the default font for the printer)</td>
</tr>
</tbody>
</table>

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:

```powerbuilder
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
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:

```powerscript
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

---

**PrintSetPrinter**

**Description**  
Sets the printer to use for the next print function call. This function does not affect open jobs.

**Syntax**  

```powerscript
PrintSetPrinter ( printername )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printername</td>
<td>String for the name of the printer you want to use</td>
</tr>
</tbody>
</table>

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

**Usage**  
The `printername` argument must use the same format as returned by the PrintGetPrinter function.

**Examples**  
This example sets the printer to the first printer in the list retrieved by the PrintGetPrinters function:

```powerscript
long ll_place
```

---

`CHAPTER 10  PowerScript Functions`
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):

```powershell
long Job

    // Define a new blank page.
    Job = PrintOpen()

    // Set the spacing factor.
    PrintSetSpacing(Job, 1.5)
```

See also
PrintOpen

---

string ls_setprn
string ls_prntrs = PrintGetPrinters ( )
ll_place=pos (ls_prntrs, "-n")
mle_1.text = PrintGetPrinters ( )
ls_setprn = Left (ls_prntrs, ll_place - 1)
PrintSetPrinter (ls_setprn)

See also
PrintGetPrinter
PrintGetPrinters
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 or if the application
user clicks Cancel in the Printer Setup dialog box.

Usage
The user’s settings have effect for the duration of the application only. After the
application exits, printer settings revert to their previous values.

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

PrintSetupPrinter

Description
Displays the printer setup dialog box

Syntax
PrintSetupPrinter()

Return value
Integer. Returns 1 if the function succeeds, 0 for cancel, -1 if an error occurs.

Usage
You can display the printer setup dialog box for different printers by first
calling the PrintSetPrinter function. You cannot change the printer by calling
PrintSetupPrinter as you can with the PrintSetup function.

Examples
This example displays the printer setup dialog box for the last printer in the list
retrieved by the PrintGetPrinters function.

long ll_place
string ls_setptr
string ls_prntrs = PrintGetPrinters()

    ll_place=lastpos (ls_prntrs, "-n")
PrintText

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
ls_setptr = Mid (ls_prntrs, ll_place + 1)
PrintSetPrinter (ls_setptr)
PrintSetupPrinter ()
```
// 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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printjobnumber</td>
<td>The number the PrintOpen function assigned to the print job</td>
</tr>
<tr>
<td>string</td>
<td>A string whose value is the text for which you want to determine the width</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns the width of string in thousandths of an inch using the current font of printjobnumber if it succeeds and -1 if an error occurs. If any argument’s value is null, PrintWidth returns null. If the returned width exceeds the maximum integer limit (+32767), PrintWidth returns -1.

**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:

```powershell
long Job
int 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")
```
CHAPTER 10  PowerScript Functions

See also  
PrintClose  
PrintOpen

**PrintX**

Description  
Reports the x coordinate of the print cursor.

Syntax  
`PrintX ( printjobnumber )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printjobnumber</code></td>
<td>The number the PrintOpen function assigned to the print job</td>
</tr>
</tbody>
</table>

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:

```powerScript
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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printjobnumber</code></td>
<td>The number the PrintOpen function assigned to the print job</td>
</tr>
</tbody>
</table>

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.
These statements print a bitmap one inch below the location of the print cursor:

```plaintext
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**

Obtains the integer value of a setting in the profile file for your application.

**Syntax**

```plaintext
ProfileInt ( filename, section, key, default )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>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.</td>
</tr>
<tr>
<td>section</td>
<td>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 <code>section</code>. Section is not case sensitive.</td>
</tr>
<tr>
<td>key</td>
<td>A string specifying the setting name in <code>section</code> whose value you want. The setting name is followed by an equal sign in the file. Do not include the equal sign in <code>key</code>. Key is not case sensitive.</td>
</tr>
<tr>
<td>default</td>
<td>An integer value that ProfileInt will return if <code>filename</code> is not found, if <code>section</code> or <code>key</code> does not exist in <code>filename</code>, or if the value of <code>key</code> cannot be converted to an integer.</td>
</tr>
</tbody>
</table>

**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 have designed for your application.

You can use SetProfileString to change values in the profile file to customize your application’s configuration at runtime. 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.
ProfileInt, ProfileString, and SetProfileString can read or write to files with ANSI or UTF16-LE encoding on Windows systems, and ANSI or UTF16-BE encoding on UNIX systems.

**Windows registry**
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 file called PROFILE.INI, which contains the following:

```
[Pb]
Maximized=1
[security]
Class=7
```

This statement returns the integer value for the keyword Maximized in section PB of file PROFILE.INI. If there were no PB section or no Maximized keyword in the PB section, it would return 3:

```
ProfileInt("C:\PROFILE.INI", "PB", "maximized", 3)
```

The following statements display a MessageBox if the integer value for the Class setting in section Security of file C:\PROFILE.INI is less than 10. The default security setting is 6 if the profile file is not found or does not 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 method for DataWindows in the DataWindow Reference or the online Help
ProfileString

Description
Obtains the string value of a setting in the profile file for your application.

Syntax
ProfileString (filename, section, key, default)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>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.</td>
</tr>
<tr>
<td>section</td>
<td>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.</td>
</tr>
<tr>
<td>key</td>
<td>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.</td>
</tr>
<tr>
<td>default</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 have designed for your application.

You can use SetProfileString to change values in the profile file to customize your application’s configuration at runtime. 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.

ProfileInt, ProfileString, and SetProfileString can read or write to files with ANSI or UTF16-LE encoding on Windows systems, and ANSI or UTF16-BE encoding on UNIX systems.

Windows registry
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 file called `PROFILE.INI`, which contains the following lines. Quotes around string values in the INI file are optional:

```plaintext
[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. In the example, the return value is Smith:

```powerscript
ProfileString("C:\PROFILE.INI", "Employee", & "Name", "None")
```

The following statements open w_marketing if the string in the keyword Name in section Department of file `C:\PROFILE.INI` is Marketing:

```powerscript
IF ProfileString("C:\PROFILE.INI", "Department", & "Name", "None") = "Marketing" THEN
    Open(w_marketing)
END IF
```

See also

ProfileInt
SetProfileString
ProfileString method for DataWindows in the *DataWindow Reference* or the online Help

---

**Rand**

Description

Obtains a random whole number between 1 and a specified upper limit.

Syntax

```powerscript
Rand (n)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>The upper limit of the range of random numbers you want returned. The lower limit is always 1. The upper limit is 32,767.</td>
</tr>
</tbody>
</table>

Return value

A numeric datatype, the datatype of `n`. Returns a random whole number between 1 and `n` inclusive. If `n` is null, Rand returns null.
Randomize

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

Randomize

Description

Initializes the random number generator so that the Rand function begins a new series of pseudorandom numbers.

Syntax

```
Randomize ( n )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read data into a string</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Read data into a character array or blob</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1

#### For reading into a string

Reads data from an OLE stream object into a string.

**Description**

Reads data from an OLE stream object into a string.

**Applies to**

OLEStream objects

**Syntax**

```
olestream.Read( variable {, stopforline } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olestream</td>
<td>The name of an OLE stream variable that has been opened.</td>
</tr>
<tr>
<td>variable</td>
<td>The name of a string variable into which want to read data from <code>olestream</code>.</td>
</tr>
<tr>
<td>stopforline (optional)</td>
<td>A boolean value that specifies whether to read a line at a time. In other words, <code>Read</code> will stop reading at the next carriage return/linefeed. Values are:</td>
</tr>
<tr>
<td></td>
<td>• 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</td>
</tr>
<tr>
<td></td>
<td>• FALSE – Read the whole stream or a maximum of 32,765 bytes</td>
</tr>
</tbody>
</table>
Read

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 does not check the functions’ return values for success, but you should be sure to check the return values in your code:

```powerbuilder
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.

Applies to
OLEStream objects
CHAPTER 10  PowerScript Functions

Syntax

```
olestream.Read ( variable {, maximumread } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olestream</td>
<td>The name of an OLE stream variable that has been opened.</td>
</tr>
<tr>
<td>variable</td>
<td>The name of a blob variable or character array into which want to read data from olestream.</td>
</tr>
<tr>
<td>maximumread</td>
<td>(optional) A long value specifying the maximum number of bytes to be read.</td>
</tr>
<tr>
<td></td>
<td>The default is 32,765 or the length of olestream.</td>
</tr>
</tbody>
</table>

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 does not 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!)
olenstr_info.Read(lb_info)
```

See also

Open
Length
Seek
Write
Real

Description

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

Syntax

Real ( stringorblob )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringorblob</td>
<td>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.</td>
</tr>
</tbody>
</table>

Return value

Real. Returns the value of stringorblob as a real. If stringorblob is not a valid PowerScript number or is an incompatible datatype, 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 method for DataWindows in the DataWindow Reference or online Help

886
CHAPTER 10  PowerScript Functions

**RecognizeText**

**Description** Specifies that text in an InkEdit control should be recognized.

**Applies to** InkEdit controls

**Syntax**

```
inkeditname.RecognizeText ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inkeditname</td>
<td>The name of the InkEdit control in which you want to recognize text.</td>
</tr>
</tbody>
</table>

**Return value** Integer. Returns 1 if text is recognized and 0 otherwise.

**Usage**

By default, ink is recognized automatically when the user pauses while entering ink and the number of milliseconds specified in the RecognitionTimer property elapses. To enable a user to pause without having text recognized, increase the RecognitionTimer interval and code the RecognizeText function in a button clicked event or another event.

**Examples**

This code in the clicked event of a “Done” button causes the recognition engine to recognize the strokes entered by the user as text:

```powerscript
boolean lb_success
lb_success = ie_1.RecognizeText()
```

---

**RegistryDelete**

**Description** Deletes a key or a value for a key in the Windows system registry.

**Syntax**

```
RegistryDelete ( key, valuename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>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.</td>
</tr>
<tr>
<td>valuename</td>
<td>A string containing the name of a value in the registry. If the specified key does not have a subkey, specifying an empty string deletes the key and its named values.</td>
</tr>
</tbody>
</table>

**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.
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")
```

See also
- RegistryGet
- RegistryKeys
- RegistrySet
- RegistryValues

**RegistryGet**

**Description**

Gets a value from the Windows system registry.

**Syntax**

```
RegistryGet( key, valuename, { valuetype }, valuevariable )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>A string whose value names the key in the system registry whose value you want.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>valuename</td>
<td>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.</td>
</tr>
<tr>
<td>valuetype</td>
<td>A value of the RegistryValue enumeration identifying the datatype of a value in the registry. Values are:</td>
</tr>
<tr>
<td></td>
<td>• RegString! – A null-terminated string</td>
</tr>
<tr>
<td></td>
<td>• RegExpandString! – A null-terminated string that contains unexpanded references to environment variables</td>
</tr>
<tr>
<td></td>
<td>• RegBinary! – Binary data</td>
</tr>
<tr>
<td></td>
<td>• ReguLong! – A 32-bit number</td>
</tr>
<tr>
<td></td>
<td>• ReguLongBigEndian! – A 32-bit number</td>
</tr>
<tr>
<td></td>
<td>• RegLink! – A Unicode symbolic link</td>
</tr>
<tr>
<td></td>
<td>• RegMultiString! – An unbounded array of strings</td>
</tr>
<tr>
<td>valuevariable</td>
<td>A variable corresponding to the datatype of valuetype in which you want to store the value obtained from the system registry for the specified key and value name.</td>
</tr>
</tbody>
</table>
Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. An error is returned if the datatype of valuevariable does not correspond to the datatype 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 more information about keys and value names in the system registry, see RegistrySet.

Examples

This statement obtains the value for the name Title and stores it in the string ls_titlefont:

```powerscript
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:

```powerscript
ulong ul_num
RegistryGet("HKEY_USERS\MyApp.Settings\Fonts", &
    "NameOfEntryNum", RegULong!, ul_num)
```

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.
RegistryKeys

Syntax

RegistryKeys ( key, subkeys )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>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.</td>
</tr>
<tr>
<td>subkeys</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```plaintext
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.

Syntax
RegistrySet (key, valuename, valuetype, value)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>A string whose value names the key in the system registry whose value you want to set.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>If key does not exist in the registry, RegistrySet creates a new key.</td>
</tr>
<tr>
<td></td>
<td>To create a key without a named value, specify an empty string for valuename.</td>
</tr>
<tr>
<td>valuename</td>
<td>A string containing the name of a value in the registry. Each key may have several named values. To specify the unnamed value, specify an empty string.</td>
</tr>
<tr>
<td></td>
<td>If valuename does not exist in the registry, RegistrySet causes a new name to be created for key.</td>
</tr>
<tr>
<td>valuetype</td>
<td>A value of the RegistryValueType enumerated datatype identifying the datatype of a value in the registry. Values are:</td>
</tr>
<tr>
<td></td>
<td>• RegString! – A null-terminated string</td>
</tr>
<tr>
<td></td>
<td>• RegExpandString! – A null-terminated string that contains unexpanded references to environment variables</td>
</tr>
<tr>
<td></td>
<td>• RegBinary! – Binary data</td>
</tr>
<tr>
<td></td>
<td>• ReguLong! – A 32-bit number</td>
</tr>
<tr>
<td></td>
<td>• ReguLongBigEndian! – A 32-bit number</td>
</tr>
<tr>
<td></td>
<td>• RegLink! – A Unicode symbolic link</td>
</tr>
<tr>
<td></td>
<td>• RegMultiString! – An unbounded array of strings</td>
</tr>
<tr>
<td>value</td>
<td>A variable corresponding to the datatype of valuetype containing a value to be set in the registry.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. An error is returned if the datatype of valuevariable does not correspond to the datatype specified in valuetype.
Long string values (more than 2048 bytes) should be stored as files and the file name stored in the registry.

### Examples

This example sets a value for the key Fonts and the value name Title:

```powerbuilder
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:

```powerbuilder
ulong ul_num
RegistrySet( &
    "HKEY_USERS\MyApp.Settings\Fonts", &
    "NameOfEntryNum", RegULong!, ul_num)
```

### See also

- RegistryDelete
- RegistryGet
- RegistryKeys
- RegistryValues
### RegistryValues

**Description**
Obtains the list of named values associated with a key.

**Syntax**
```
RegistryValues ( key, valuename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>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.</td>
</tr>
<tr>
<td>valuename</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 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.
**RelativeDate**

**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 2006-02-10:

RelativeDate(2006-01-31, 10)

This statement returns 2006-01-21:

RelativeDate(2006-01-31, -10)

**See also**

DaysAfter
RelativeDate method for DataWindows in the DataWindow Reference or the online Help

---

**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.

**Usage**

All PowerBuilder functions except RelativeTime use the Windows API to process dates and times, so in most instances processing of Daylight Savings Time (DST) is dependent on the operating system. However, the RelativeTime function is not DST-aware and therefore may return an incorrect time in a region that is using DST.
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 method for DataWindows in the DataWindow Reference or the online Help

---

**ReleaseAutomationNativePointer**

**Description**
Releases the pointer to an OLE object that you got with GetAutomationNativePointer.

**Applies to**
OLEObject

**Syntax**

```
oleobject.ReleaseAutomationNativePointer ( pointer )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>The name of an OLEObject variable containing the object for which you want to release the native pointer.</td>
</tr>
<tr>
<td>pointer</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 0 if it succeeds and -1 if an error occurs.

**Usage**

`Pointer` is a pointer to OLE’s IUnknown interface. You can use IUnknown::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

---

PowerScript Reference 895
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
Pointer is a pointer to OLE’s IUnknown interface. You can use IUnknown::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

RemoveDirectory

RemoveDirectory

Description
Removes a directory.

Syntax
RemoveDirectory ( directoryname )

Argument | Description
---|---
directoryname | String for the name of the directory you want to remove. If you do not specify an absolute path, this function deletes relative to the current working directory.

Return value
Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Usage
The directory must be empty and must not be the current directory for this function to succeed.
Examples
This example removes a subdirectory from the current directory:

```powerscript
string ls_path="my targets"
integer li_filename

li_filename = RemoveDirectory ( ls_path )
If li_filename <> 1 then
    MessageBox("Remove directory failed", &
        + "Check that the directory exists, is empty, and " &
        + "is not the current directory")
else
    MessageBox("Success", "Directory " + ls_path + &
        " deleted")
end if
```

See also
DirectoryExists
GetCurrentDirectory

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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipelineobject</td>
<td>The name of a pipeline user object that contains the pipeline object being executed</td>
</tr>
<tr>
<td>destinationtrans</td>
<td>The name of a transaction object with which to connect to the target database</td>
</tr>
</tbody>
</table>

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:

```powerbuilder
i_pipe.Replace(i_dst)
```

**See also**

Cancel

Repair

Start

---

**Replace**

**Description**

Replaces a portion of one string with another.

**Syntax**

```
Replace ( string1, start, n, string2 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string1</code></td>
<td>The string in which you want to replace characters with <code>string2</code>.</td>
</tr>
<tr>
<td><code>start</code></td>
<td>A long whose value is the number of the first character you want replaced. (The first character in the string is number 1.)</td>
</tr>
<tr>
<td><code>n</code></td>
<td>A long whose value is the number of characters you want to replace.</td>
</tr>
<tr>
<td><code>string2</code></td>
<td>The string that will replace characters in <code>string1</code>. The number of characters in <code>string2</code> can be greater than, equal to, or less than the number of characters you are replacing.</td>
</tr>
</tbody>
</table>

**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:

```powerscript
string Name
Name = "Davis"
Name = Replace(Name, 4, 2, "e")
```

This statement returns BABY RUTH:

```powerscript
Replace("BABE RUTH", 1, 4, "BABY")
```

This statement returns Closed for the Winter:

```powerscript
Replace("Closed for Vacation", 12, 8, "the Winter")
```

This statement returns ABZZZZEF:

```powerscript
Replace("ABCDEF", 3, 2, "ZZZZ")
```

This statement returns ABZZZZ:

```powerscript
Replace("ABCDEF", 3, 50, "ZZZZ")
```

This statement returns ABCDEFZZZZ:

```powerscript
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:

```powerscript
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))
LOOPLoop
    sle_1.Text = mystring
See also Replace method for DataWindows in the DataWindow Reference or the online Help

ReplaceA

Description
Temporarily converts a string to DBCS based on the current locale, then replaces a portion of one string with another.

Syntax
ReplaceA (string1, start, n, string2)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td>The string containing characters you want to replace.</td>
</tr>
<tr>
<td>start</td>
<td>A long whose value is the position in bytes of the first character you want to replace in string1.</td>
</tr>
<tr>
<td>n</td>
<td>A long whose value is the number of bytes you want to replace in string1.</td>
</tr>
<tr>
<td>string2</td>
<td>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.</td>
</tr>
</tbody>
</table>

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, ReplaceA returns null.

Usage
ReplaceA replaces the functionality that Replace had in DBCS environments in PowerBuilder 9. ReplaceA replaces a string by number of bytes, whereas Replace replaces a string by number of characters in both SBCS and DBCS environments. ReplaceA also specifies the starting position of the string to be replaced by number of bytes, whereas Replace specifies the starting position by number of characters.

In SBCS environments, Replace, ReplaceW, and ReplaceA return the same results.
ReplaceText

Replaces selected text in an edit control with a specified string.

**Applies to**
DataWindow, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, and DropDownPictureListBox controls

**Syntax**
```
editname.ReplaceText (string)
```

**Argument | Description**
---|---
editname | The name of the DataWindow, EditMask, InkEdit, 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**
Integer for DataWindow, InkEdit, and list boxes, Long for other controls.

For InkEdit, returns 1 for success and -1 if an error occurs. For other controls, 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")
```
ReplaceW

Description
Replaces a portion of one string with another. This function is obsolete. It has the same behavior as Replace in all environments.

Syntax
ReplaceW( string1, start, n, string2 )

Reset
Clears data from a control or object. The syntax you choose depends on the target object.

For syntax for DataWindows and DataStores see the Reset method for DataWindows in the DataWindow Reference or the online Help.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete all items from a list</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Delete all the data (and optionally the series and categories) from a graph</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Return to the beginning of a trace file</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

Syntax 1

For list boxes

Description
Deletes all the items from a list.

Applies to
ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax
listboxname.Reset( )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>The name of the ListBox control from which to delete all items</td>
</tr>
</tbody>
</table>

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 ddb_Actions:

```powerscript
ddlb_Actions.Reset()
```

See also

DeleteItem

**Syntax 2 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**

```powerscript
controlname.Reset(graphresettype)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the graph object in which you want to delete all the data values or all series and all data values</td>
</tr>
<tr>
<td><code>graphresettype</code></td>
<td>A value of the grResetType enumerated datatype specifying whether you want to delete only data values or all series and all data values:</td>
</tr>
<tr>
<td></td>
<td>• All! – Delete all series, categories, and data in <code>controlname</code></td>
</tr>
<tr>
<td></td>
<td>• Category! – Delete categories and data in <code>controlname</code></td>
</tr>
<tr>
<td></td>
<td>• Data! – Delete data in <code>controlname</code></td>
</tr>
<tr>
<td></td>
<td>• Series! – Delete the series and data in <code>controlname</code></td>
</tr>
</tbody>
</table>

**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`:

```powerscript
gr_product_data.Reset(Series!)
```

See also

AddData
AddSeries
**Reset**

### Syntax 3

**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

- **FileNotFoundException!** – 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 execution to the start of the open trace file `ltf_file` so that the file’s contents can be reread:

```powerbuilder
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:

```javascript
... 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:

```vbnet
... retcd = PBRX1.TriggerPBEvent(theEvent, numargs)
rc = PBRX1.GetLastReturn()
IF rc <> 1 THEN
  messagebox "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

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 containing the graph
graphcontrol (DataWindow control only) | (Optional) A string whose value is the name of the graph in the DataWindow control in which you want to reset the color
seriesnumber | The number of the series in which you want to reset the color of a data point
datapointnumber | The number of the data point for which you want to reset the color

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

```powerbuilder
SeriesNbr = gr_product_data.FindSeries("Costs")
gr_product_data.ResetDataColors (SeriesNbr, 10)
```

These statements change the color of data point 10 in the series named Costs in the graph gr_comps in the DataWindow control dw_equip to the color for the series:

```powerbuilder
SeriesNbr = dw_equipment.FindSeries("Costs")
dw_equip.ResetDataColors ("gr_comps", SeriesNbr, 10)
```

See also
GetDataStyle
SeriesName
GetSeriesStyle
SetDataStyle
SetSeriesStyle

906 PowerBuilder
ResetInk

Description: Clears ink from an InkPicture control.

Applies to: InkPicture controls

Syntax:

\[ \text{inkpicname}.\text{ResetInk}() \]

Argument Description
---
inkpicname The name of the InkPicture control from which you want to clear ink.

Return value: Integer. Returns 1 for success and -1 for failure.

Usage: Use the ResetInk function to clear the ink from an InkPicture control.

Examples: The following example clears the ink from an InkPicture control:

\[ \text{ip}_1.\text{ResetInk}() \]

See also: LoadInk, LoadPicture, ResetPicture, SaveInk, Save

ResetPicture

Description: Clears a picture from an InkPicture control.

Applies to: InkPicture controls

Syntax:

\[ \text{inkpicname}.\text{ResetPicture}() \]

Argument Description
---
inkpicname The name of the InkPicture control from which you want to clear a picture.

Return value: Integer. Returns 1 for success and -1 for failure.

Usage: Use the ResetInk function to clear the image from an InkPicture control.

Examples: The following example clears the image from an InkPicture control:

\[ \text{ip}_1.\text{ResetPicture}() \]
**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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectname</td>
<td>The name of the object or control you want to resize</td>
</tr>
<tr>
<td>width</td>
<td>The new width in PowerBuilder units</td>
</tr>
<tr>
<td>height</td>
<td>The new height in PowerBuilder units</td>
</tr>
</tbody>
</table>

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
```

The first two statements, although they redraw `gb_box1` twice, achieve the same result as the third statement:

```
gb_box1.Width = 100 // These lines resize
gb_box1.Height = 150 // gb_box1 to 100 x 150
gb_box1.Resize(100, 150)// So does this line
```
Examples

This statement changes the Width and Height properties of gb_box1 and redraws gb_box1 with the new properties:

\[ gb\_box1.\text{Resize}(100, 150) \]

This statement doubles the width and height of the picture control p_1:

\[ p\_1.\text{Resize}(p\_1.\text{Width}\times2, p\_1.\text{Height}\times2) \]

---

**Resolve_Initial_References**

**Description**

Uses the CORBA naming service API to obtain the initial naming context for an EAServer component.

This function is used by PowerBuilder clients connecting to EAServer.

**Applies to**

JaguarORB objects

**Syntax**

\[ jaguarorb.\text{Resolve}_\text{Initial}_\text{References} (\text{objstring}, \text{object}) \]

**Argument** | **Description**
--- | ---
\(jaguarorb\) | An instance of JaguarORB
\(\text{objstring}\) | A string that has the value "NameService"
\(\text{object}\) | A reference variable of type CORBAobject that will contain a reference to the COS naming service

**Return value**

Long. Returns 0 if it succeeds and a negative number if an error occurs.

**Usage**

If you want to use the Jaguar naming service API, you can use the \(\text{Resolve}_\text{Initial}_\text{References}\) function to obtain the initial naming context. However, this technique is not recommended because it requires use of a deprecated \(\text{SessionManager}:\text{Factory}\)\text{create} method. Most PowerBuilder clients do not need to use the CORBA naming service explicitly. Instead, they can rely on the name resolution that is performed automatically when they create EAServer component instances using the \(\text{CreateInstance}\) and \(\text{Lookup}\) functions of the Connection object.

You can also use the JaguarORB object’s \(\text{String} \_\text{To} \_\text{Object}\) function to instantiate a proxy instance without using the CORBA naming service explicitly. For more information about connecting to EAServer using the JaguarORB object, see *Application Techniques*.

When you use the CORBA naming service, you need to generate proxies for the naming service interface and include these proxies in the library list for the client.
Resolve_Initial_References

Examples

The following example shows the use of the Resolve_Initial_References function to obtain an initial naming context. After obtaining the naming context, it uses the naming context’s resolve method to obtain a reference to a Factory object for the component and then narrows that reference to the SessionManager’s Factory interface.

The resolve method takes a name parameter, which is a sequence of NameComponent structures. Each NameComponent structure has an id attribute that identifies the component and a kind attribute that can be used to describe the component. In the example below, the name has only one component. The create method of the Factory object obtains proxies for the component. It returns a CORBA object reference that you can convert into a reference to the component’s interface using the Narrow function.

The NamingContext and NameComponent types used in the example are proxies imported from the CosNaming package in EAServer, and the Factory type is imported from the SessionManager package:

```java
CORBAObject my_corbaobj
JaguarORB my_orb
NamingContext my_nc
NameComponent the_name[]
Factory my_Factory
n_jagcomp my_jagcomp

my_orb = CREATE JaguarORB
// Enclose the name of the URL in single quotes
my_orb.init("ORBNameServiceURL='iiop://server1:2000'")

my_orb.Resolve_Initial_References("NameService", &
    my_corbaobj)
my_corbaobj._narrow(my_nc, &
    "omg.org/CosNaming/NamingContext")

the_name[1].id = "mypackage/n_jagcomp"
the_name[1].kind = ""

TRY
    my_corbaobj = my_nc.resolve(the_name)
    my_corbaobj._narrow(my_Factory, &
        "SessionManager/Factory")
    my_corbaobj = my_Factory.create("jagadmin",""
    my_corbaobj._narrow(my_jagcomp, &
        "mypackage/n_jagcomp")
CATCH (Exception e)
    MessageBox("Exception Raised!", e.getMessage())
```
END TRY
my_jagcomp.getdata()

See also
Init
_Narrow
String_To_Object

RespondRemote

Description
Sends a DDE message indicating whether the command or data received from a remote DDE application was acceptable.

Syntax
\[ \text{RespondRemote(} \text{boolean} \text{)} \]

Argument | Description
---|---
\text{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 \text{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 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:

\[ \text{String Applname, Topic, Item, Value} \]
\[ \text{GetDataDDEOrigin(Applname, Topic, Item)} \]
\[ \text{IF Applname = "Gateway" THEN} \]
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 137

ResumeTransaction

Description
Associates the EA.server transaction passed as an argument with the calling thread.

Applies to
CORBACurrent objects
CHAPTER 10  PowerScript Functions

Syntax

```
CORBACurrent.ResumeTransaction ( handletrans )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBACurrent</td>
<td>Reference to the CORBACurrent service instance</td>
</tr>
<tr>
<td>handletrans</td>
<td>An unsignedlong containing the handle of a suspended transaction</td>
</tr>
</tbody>
</table>

Return value

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

-1 Unknown failure

-2 The transaction referred to by handletrans is no longer valid

Usage

The ResumeTransaction function associates the transaction referred to by handletrans argument with the calling thread. The argument is obtained from a call to SuspendTransaction and may refer to a transaction that was previously associated with the current thread or with a different thread in the same execution environment.

Caution

The handletrans argument must be obtained from the SuspendTransaction function. Using any other value as the argument to ResumeTransaction may have unpredictable results.

ResumeTransaction can be called by a client or a component that is marked as OTS style. must be using the two-phase commit transaction coordinator (OTS/XA).

Examples

This example shows the use of the ResumeTransaction function to associate the calling thread with the transaction referred to by the l.handle argument returned by SuspendTransaction:

```powerScript
// Instance variable:
// CORBACurrent corbcurr
integer li_rc
unsignedlong l1_handle

li_rc = this.GetContextService("CORBACurrent", & corbcurr)
li_rc = corbcurr.Init()
li_rc = corbcurr.BeginTransaction()
// do some transactional work
l1_handle = corbcurr.SuspendTransaction()
// do some non-transactional work
li_rc = corbcurr.ResumeTransaction(l1_handle)
```
Reverse

// do some more transactional work
li_rc = corbcurr.CommitTransaction()

See also
BeginTransaction
CommitTransaction
GetContextService
GetStatus
GetTransactionName
Init
RollbackOnly
RollbackTransaction
SetTimeout
SuspendTransaction

Reverse

Description
Reverses the order or characters in a string.

Syntax
Reverse ( string )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string whose characters you want to reorder so that the last character is first and the first character is last</td>
</tr>
</tbody>
</table>

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 right-to-left character display when you are using a version of Windows that supports right-to-left languages.

Examples
Under a a version of Windows that supports right-to-left languages, 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
RevertToSelf

Description
Restores the security attributes for a COM object that is running on COM+ and impersonating the client.

Applies to
TransactionServer objects

Syntax
transactionserver.RevertToSelf ( )

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

Usage
COM objects running on COM+ can use the ImpersonateClient function to run in the client’s security context so that the object has access to the same resources as the client. Use RevertToSelf to restore the object’s security context.

Examples
The following example creates an instance of the TransactionServer service and checks whether the COM object is currently running in the client’s security context. If it is, it reverts to the object’s security context:

```powerShell
TransactionServer txninfo_test
integer li_rc
li_rc = GetContextService( "TransactionServer", &
   txninfo_test )
IF  txninfo_test.IsImpersonating() THEN  &
   txninfo_test.RevertToSelf()
```

See also
ImpersonateClient
IsCallerInRole
IsImpersonating
IsSecurityEnabled
**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 )
```

**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.

<table>
<thead>
<tr>
<th>Color</th>
<th>Red value</th>
<th>Green value</th>
<th>Blue value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>White</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Light Gray</td>
<td>192</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>Dark Gray</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Red</td>
<td>255</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dark Red</td>
<td>128</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Examples

This statement returns a long that represents black:

\[
\text{RGB}(0, 0, 0)
\]

This statement returns a long that represents white:

\[
\text{RGB}(255, 255, 255)
\]

These statements set the color properties of the StaticText \texttt{st\_title} to be green letters on a dark magenta background:

\[
\begin{align*}
st\_title.\text{TextColor} &= \text{RGB}(0, 255, 0) \\
st\_title.\text{BackColor} &= \text{RGB}(128, 0, 128)
\end{align*}
\]

See also

RGB method for DataWindows in the \textit{DataWindow Reference} or the online Help

---

**Right**

Description

Obtains a specified number of characters from the end of a string.

Syntax

\[
\text{Right}(\text{string}, n)
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{string}</td>
<td>The string from which you want characters returned</td>
</tr>
<tr>
<td>\textit{n}</td>
<td>A long whose value is the number of characters you want returned from the right end of \textit{string}</td>
</tr>
</tbody>
</table>
RightA

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 method for DataWindows in the DataWindow Reference or online Help

RightA

Description
Temporarily converts a string from Unicode to DBCS based on the current locale, then returns the specified number of bytes from the end of the string.

Syntax
RightA ($string$, $n$)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$string$</td>
<td>The string you want to search</td>
</tr>
<tr>
<td>$n$</td>
<td>A long whose value is the number of bytes you want returned from the right end of $string$</td>
</tr>
</tbody>
</table>

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, RightA returns null. If $n$ is greater than or equal to the length of the string, RightA returns the entire string. It does not add spaces to make the return value’s length equal to $n$.

Usage
RightA replaces the functionality that Right had in DBCS environments in PowerBuilder 9.

In SBCS environments, Right, RightW, and RightA return the same results.
CHAPTER 10  PowerScript Functions

RightW

Description
Obtains a specified number of characters from the end of a string. This function is obsolete. It has the same behavior as Right in all environments.

Syntax
RightW ( string, n )

RightTrim

Description
Removes spaces from the end of a string.

Syntax
RightTrim ( string {, removeallspaces } )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string you want returned with trailing blanks deleted</td>
</tr>
<tr>
<td>removeallspaces</td>
<td>A boolean indicating that all types of spaces should be deleted</td>
</tr>
</tbody>
</table>

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.

Usage
If you do not include the optional removeallspaces argument or its value is false, only the space character (U+0020) is removed from the string.

If the removeallspaces argument is set to true, all types of space characters are removed. See LeftTrim for a list of space characters.

Examples
This statement returns RUTH if all the trailing blanks are space characters:

RightTrim("RUTH ")

This statement returns RUTH if the trailing blanks include other types of white space characters:

RightTrim("RUTH ", true)

See also
LeftTrim
Trim
RightTrim method for DataWindows in the DataWindow Reference or the online Help
RightTrimW

**Description**
Removes spaces from the end of a string. This function is obsolete. It has the same behavior as RightTrim in all environments.

**Syntax**
```plaintext
RightTrimW ( string )
```

RollbackOnly

**Description**
Modifies an EAServer transaction associated with a calling thread so that the only possible outcome is to roll back the transaction.

** Applies to **
CORBACurrent objects

**Syntax**
```plaintext
CORBACurrent.RollbackOnly ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBACurrent</td>
<td>Reference to the CORBACurrent service instance</td>
</tr>
</tbody>
</table>

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

-1 Failed for unknown reason
-2 No transaction is associated with the calling thread

**Usage**
RollbackTransaction is typically called by the originator of the transaction. Another participant in a client- or OTS style transaction can call RollbackOnly to vote that the transaction should be rolled back.

RollbackOnly can be called by a client or a component that is marked as OTS style. EAServer must be using the two-phase commit transaction coordinator (OTS/XA).

**Examples**
In this example, a participant in a transaction has determined that it should be rolled back. It creates and initializes an instance of the CORBACurrent service object and votes to roll back the transaction:

```plaintext
// Instance variable:
// CORBACurrent corbcurr
int li_rc

li_rc = this.GetContextService("CORBACurrent", & corbcurr)
IF li_rc <> 1 THEN
   // handle the error
```
END IF

li_rc = corbcurr.Init()
IF li_rc <> 0 THEN
  // handle the error
ELSE
  corbcurr.RollbackOnly()
END IF

See also
BeginTransaction
CommitTransaction
GetContextService
GetStatus
GetTransactionName
Init
ResumeTransaction
RollbackTransaction
SetTimeout
SuspendTransaction

RollbackTransaction

Description
Rolls back the EAServer transaction associated with the calling thread.

Applies to
CORBACurrent objects

Syntax
CORBACurrent.RollbackTransaction()  

Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBACurrent</td>
<td>Reference to the CORBACurrent service instance</td>
</tr>
</tbody>
</table>

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

-1 Failed for unknown reason
-2 No transaction is associated with the calling thread
-3 The calling thread does not have permission to commit the transaction
-4 The HeuristicCommit exception was raised
**RollbackTransaction**

**Usage**

The `RollbackTransaction` function rolls back the transaction associated with the calling thread. The call fails if the HeuristicCommit exception is raised. Heuristic decisions are usually made when normal processing cannot continue, such as when a communications failure occurs. The HeuristicCommit exception is raised when all relevant updates have been committed.

`RollbackTransaction` can be called by a client or a component that is marked as OTS style. EAServer must be using the two-phase commit transaction coordinator (OTS/XA).

**Examples**

This example shows the use of `RollbackTransaction` to roll back a transaction when an update does not succeed:

```powershell
// Instance variables:
// CORBACurrent corbcurr
int li_rc1, li_rc2
long ll_rc

done.GetContextService("CORBACurrent", corbcurr)
li_rc1 = corbcurr.Init()
IF li_rc1 <> 1 THEN
    // handle the error
ELSE
    ll_rc = CreateInstance(mycomp)
    // invoke methods on the instantiated component
    // test return values and roll back
    // if unsatisfactory

    IF li_rc2 = 1 THEN
        corbcurr.CommitTransaction()
    ELSE
        corbcurr.RollbackTransaction()
    END IF
END IF
```

**See also**

- `BeginTransaction`
- `CommitTransaction`
- `GetContextService`
- `GetStatus`
- `GetTransactionName`
- `Init`
- `ResumeTransaction`
- `RollbackOnly`
- `SetTimeout`
- `SuspendTransaction`
**Round**

**Description**
Rounds a number to the specified number of decimal places.

**Syntax**
```
Round(x, n)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>The number you want to round.</td>
</tr>
<tr>
<td>n</td>
<td>The number of decimal places to which you want to round x. Valid values are 0 through 30.</td>
</tr>
</tbody>
</table>

**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:
  ```PowerScript
  Round(9.624, 2)
  ```
- This statement returns 9.63:
  ```PowerScript
  Round(9.625, 2)
  ```
- This statement returns 9.600:
  ```PowerScript
  Round(9.6, 3)
  ```
- This statement returns -9.63:
  ```PowerScript
  Round(-9.625, 2)
  ```
- This statement returns null:
  ```PowerScript
  Round(-9.625, -1)
  ```

**See also**
- Ceiling
- Int
- Truncate
- Round method for DataWindows in the *DataWindow Reference* or the online Help
RoutineList

Description Provides a list of the routines included in a performance analysis model.

Applies to ProfileClass and Profiling objects

Syntax

`instancename.RoutineList( list )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instancename</code></td>
<td>Instance name of the ProfileClass or Profiling object.</td>
</tr>
<tr>
<td><code>list</code></td>
<td>An unbounded array variable of datatype 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.</td>
</tr>
</tbody>
</table>

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:

```pascal
Long ll_cnt
ProfileCall lproc_call[]

lpro_model.BuildModel() 
lpro_model.RoutineList(iprot_list) 
... 
```

See also ClassList
CHAPTER 10  PowerScript Functions

Run

Description
Runs the specified application program.

Syntax
Run ( string {, windowstate } )

Argument | Description
--- | ---
string | A string whose value is the file name 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 datatype 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
- 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 file name, 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 Clock accessory application in its normal size:

Run("Clock")

This statement runs the Microsoft Windows Clock accessory application minimized:

Run("Clock", Minimized!)

This statement runs the program WINNER.COM on the C drive in a maximized state. The parameter passed to WINNER.COM opens 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
Save

Saves a picture and optionally overlay ink to a file or blob from an InkPicture control or saves an OLE object in an OLE control or an OLE storage object. The syntax you use depends on the type of object you want to save.

<table>
<thead>
<tr>
<th>To</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save the contents of an InkPicture control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Save an OLE object</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1

**For InkPicture controls**

**Description**
Saves a picture and optionally overlay ink to a file or blob from an InkPicture control.

**Applies to**
InkPicture controls

**Syntax**

```powerscript
inkpicname.Save(t | b, format {, WithInk } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inkpicname</code></td>
<td>The name of the InkPicture control from which you want to save a picture.</td>
</tr>
<tr>
<td><code>t</code></td>
<td>A string containing the name and location of the file into which the picture will be saved.</td>
</tr>
<tr>
<td><code>b</code></td>
<td>The name of a blob passed by reference that will hold the picture in the control.</td>
</tr>
<tr>
<td><code>format</code></td>
<td>An integer specifying the format in which the picture is to be saved. Values are: 0 – BMP (bitmap) 1 – JPEG (Joint Photographic Experts Group) 2 – GIF (Graphics Interchange Format) 3 – TIFF (Tagged Image File Format) 4 – PNG (Portable Network Graphics)</td>
</tr>
<tr>
<td><code>WithInk</code> (optional)</td>
<td>A boolean specifying whether overlay ink should be saved with the picture. Values are: <code>True</code> – overlay ink is saved with the picture (default) <code>False</code> – overlay ink is not saved with the picture</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 1 for success and -1 for failure.

**Usage**
Use the `Save` function to save the image in an InkPicture control to a file or blob with or without any ink annotations that have been made to it. By default, the ink is saved with the image.
Save

Examples

The following example saves the image in an InkPicture control and its ink annotations in bitmap format into a blob, and attempts to update the image in the database:

```pascal
int li_return
blob lblb_ink

li_return = ip_1.save(lblb_ink, 0, true)

UPDATEBLOB employee SET backimage = :lbb_ink WHERE emp_id = :gi_id;

IF sqlca.SQLNRows > 0 THEN
  COMMIT;
ELSE
  messagebox("Update failed",sqlca.sqlerrtext)
END IF
```

The following example saves the image in an InkControl into a GIF file without any ink annotations:

```pascal
int li_return
string ls_pathname, ls_filename

GetFileSaveName("Save As", ls_pathname, ls_filename, "GIF")
li_return = ip_1.save(ls_pathname, 2, false)
```

See also
LoadInk
LoadPicture
ResetInk
ResetPicture
SaveInk

Syntax 2

**For OLE objects**

Saves an OLE object in an OLE control or an OLE storage object.

**Syntax**

```
oleobject.Save()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>The name of an OLE control or an OLE storage variable</td>
</tr>
</tbody>
</table>
### 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 returns 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 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:

```powerbuilder
integer result
result = ole_1.Save()
```

This example saves a storage object to its file. `Olestor_1` is an instance variable of type `olestorage`:

```powerbuilder
integer result
result = olestor_1.Save()
```
SaveAs

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:

```powershell
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:

```powershell
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.

For DataWindow and DataStore syntax, see the SaveAs method for DataWindows in the DataWindow Reference or the online Help.

<table>
<thead>
<tr>
<th>To</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save the data in a graph</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Save the OLE object in an OLE control to a storage file</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Save the OLE object in an OLE control to a storage object in memory</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>Save an OLE storage and any controls that have opened that storage in a file</td>
<td>Syntax 4</td>
</tr>
<tr>
<td>Save an OLE storage object in another OLE storage object</td>
<td>Syntax 5</td>
</tr>
</tbody>
</table>
Syntax 1  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  

\[ \text{controlname}.\text{SaveAs} ( \{ \text{filename}, \} \{ \text{graphcontrol}, \text{saveastype}, \text{colheading} \}, \text{encoding} ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph control whose contents you want to save or the name of the DataWindow DataStore containing the graph.</td>
</tr>
<tr>
<td>filename</td>
<td>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 (&quot;&quot;), PowerBuilder prompts the user for a file name.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>A string whose value is the name of the graph in the DataWindow control or DataStore whose contents you want to save.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
</tr>
</tbody>
</table>
### SaveAs

**Argument** | **Description**
---|---
`saveastype` (optional) | A value of the `SaveAsType` enumerated datatype 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
- `EMF` – Enhanced Metafile format
- `Excel` – Microsoft Excel format
- `Excel5` – Microsoft Excel version 5 format
- `Excel8` – Microsoft Excel version 8 and higher format
- `HTMLTable` – HTML TABLE, TR, and TD elements
- `PDF` – Adobe Portable Document 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
- `XML` – Extensible Markup Language
- `XSLFO` – Extensible Stylesheet Language Formatting Objects

**Obsolete values**
The following `SaveAsType` values are considered to be obsolete and will be removed in a future release: `Excel`, `WK1`, `WKS`, `SYLK`, `dBase2`, `WMF`. Use `Excel8` for current versions of Microsoft Excel and `EMF` in place of `WMF`.

**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.
CHAPTER 10  PowerScript Functions

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
You must use zero or three arguments. If you do not specify any arguments for SaveAs, PowerBuilder displays the Save As dialog box, letting the user specify the format of the saved data.

Regional settings
If you use date formats in your graph, you must verify that yyyy is the Short Date Style for year in the Regional Settings of the user’s Control Panel. Your program can check this with the RegistryGet function.

If the setting is not correct, you can ask the user to change it manually or to have the application change it (by calling the RegistrySet function). The user may need to reboot after the setting is changed.

Examples
This statement saves the contents of the graph gr_History. The file and format information are not specified, so PowerBuilder prompts for the file name and save the graph as tab-delimited text:

```
gr_History.SaveAs()
```

This statement saves the contents of gr_History to the file G:\HR\EMPLOYEE.HIS. The format is CSV without column headings:

```
gr_History.SaveAs("G:\HR\EMPLOYEE.HIS", CSV!, FALSE)
```

This statement saves the contents of gr_computers in the DataWindow control dw_equipmt to the file G:\INVENTORY\SALES.XLS. The format is Excel with column headings:

```
dw_equipmt.SaveAs("gr_computers", "G:\INVENTORY\SALES.XLS", Excel!, TRUE)
```

See also
Print

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoding (optional)</td>
<td>Character encoding of the file to which the data is saved. This parameter applies only to the following formats: TEXT, CSV, SQL, HTML, and DIF. If you do not specify an encoding parameter, the file is saved in ANSI format. Values are:</td>
</tr>
<tr>
<td></td>
<td>- EncodingANSI! (default)</td>
</tr>
<tr>
<td></td>
<td>- EncodingUTF8!</td>
</tr>
<tr>
<td></td>
<td>- EncodingUTF16LE!</td>
</tr>
<tr>
<td></td>
<td>- EncodingUTF16BE!</td>
</tr>
</tbody>
</table>
Syntax 2

For saving an OLE control to a file

Description
Saves the object in an OLE control in a storage file.

Applies to
OLE controls

Syntax

\texttt{olecontrol.SaveAs(OLEtargetfile)}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{olecontrol}</td>
<td>The name of the OLE control containing the object you want to save.</td>
</tr>
<tr>
<td>\texttt{OLEtargetfile}</td>
<td>A string specifying the name of an OLE storage file. The file can already exist. \texttt{OLEtargetfile} can include a path, as well as information about where to store the object in the file's internal structure.</td>
</tr>
</tbody>
</table>

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, \texttt{SaveAs} returns null.

Usage
The \texttt{Open} function establishes a connection between a storage file and a storage object, or a storage file or object and an OLE control. The \texttt{Save} function uses this connection to save the OLE data.

When you call \texttt{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 \texttt{Save} function.

Examples
This example saves the object in the control \texttt{ole_1}:

\begin{verbatim}
integer result
result = ole_1.SaveAs("c:\ole\expense.ole")
\end{verbatim}

See also
Open
Save
### Syntax 3

**For saving an OLE control to an OLE storage**

**Description**
Saves the object in an OLE control to an OLE storage object in memory.

**Applies to**
OLE controls

**Syntax**

```powerscript
olecontrol.SaveAs (targetstorage, substoragename)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olecontrol</td>
<td>The name of the OLE control containing the object you want to save.</td>
</tr>
<tr>
<td>targetstorage</td>
<td>The name of an object variable of OLEStorage in which to store the object in olecontrol.</td>
</tr>
<tr>
<td>substoragename</td>
<td>A string whose value is the name of a substorage within targetstorage. If substorage does not exist, SaveAs creates it.</td>
</tr>
</tbody>
</table>

**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`:

```powerscript
integer result
result = ole_1.SaveAs(stg_stuff)
```

**See also**
Open  
Save
SaveAs

Syntax 4

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.

Applies to
OLE storage objects

Syntax

olestorage.SaveAs (OLEtargetfile )

Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olestorage</td>
<td>The name of an object variable of type OLEStorage containing the OLE object you want to save.</td>
</tr>
<tr>
<td>OLEtargetfile</td>
<td>A string specifying the name of a new OLE storage file. OLEtargetfile can include a path.</td>
</tr>
</tbody>
</table>

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 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:

```powershell
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:

```powerscript
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 5  For saving an OLE storage object in another OLE storage

Description
Saves an OLE storage object to another OLE storage object variable in memory.

Applies to
OLE storage objects

Syntax
`olestorage.SaveAs ( substoragename, targetstorage )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olestorage</td>
<td>The name of an object variable of type OLEStorage containing the OLE object you want to save.</td>
</tr>
<tr>
<td>substoragename</td>
<td>A string whose value is the name of a substorage within targetstorage. If substorage does not exist, SaveAs creates it.</td>
</tr>
<tr>
<td>targetstorage</td>
<td>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.</td>
</tr>
</tbody>
</table>
SaveDocument

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 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:

```pascal
integer result
result = stg_stuff.SaveAs("copy1", stg_clone)
```

See also

Close
Open
Save

SaveDocument

Description

Saves the contents of a RichTextEdit control in a file. You can specify either rich-text format (RTF) or text format for the file.

Applies to

RichTextEdit controls
Syntax

\[ \text{rtename.SaveDocument} ( \text{filename}, \text{filetype}, \text{encoding} ) ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control whose contents you want to save.</td>
</tr>
<tr>
<td>filename</td>
<td>A string whose value is the name of the file to be saved. If the file already exists, the FileExists event is triggered.</td>
</tr>
<tr>
<td>filetype</td>
<td>A value of the FileType enumerated datatype specifying the format of the saved file. Values are:</td>
</tr>
<tr>
<td></td>
<td>• FileTypeRichText! – Save the file in rich text format</td>
</tr>
<tr>
<td></td>
<td>• FileTypeText! – Save the file as text</td>
</tr>
<tr>
<td></td>
<td>• FileTypeDoc! – Save the file in Microsoft Word format</td>
</tr>
<tr>
<td></td>
<td>• FileTypeHTML! – Save the file in HTML format</td>
</tr>
<tr>
<td></td>
<td>• FileTypePDF! – Save the file in PDF format</td>
</tr>
<tr>
<td>encoding</td>
<td>Character encoding of the file to which the data is saved. This parameter applies only to text files. If you do not specify an \text{encoding} parameter, the file is saved in ANSI format.</td>
</tr>
<tr>
<td></td>
<td>The \text{filetype} argument must be set to FileTypeText! If the \text{filetype} argument is set to any other file type, this argument is ignored.</td>
</tr>
<tr>
<td></td>
<td>Values are:</td>
</tr>
<tr>
<td></td>
<td>• EncodingANSI! (default)</td>
</tr>
<tr>
<td></td>
<td>• EncodingUTF8!</td>
</tr>
<tr>
<td></td>
<td>• EncodingUTF16LE!</td>
</tr>
<tr>
<td></td>
<td>• EncodingUTF16BE!</td>
</tr>
</tbody>
</table>

Return value

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

Usage

\text{SaveDocument} triggers a FileExists event when the file you name already exists. If you do not specify a \text{filetype}, \text{SaveDocument} saves the file as a text file if you specify a file name with the extension .txt, as a Microsoft Word document if you specify a file name with the extension .doc, and as an RTF file if you specify a file name with the .rtf extension.

The format that you specify in the \text{encoding} argument is valid only if you specified FileTypeText! for the \text{filetype} argument. \text{SaveDocument} saves text in ANSI format only for all other file types.

Examples

This code for a CommandButton saves the document in the RichTextEdit \text{rte_1}:

\[
\begin{verbatim}
integer li_rtn
li_rtn = rte_1.SaveDocument("c:\test.rtf", & FileTypeRichText!)
\end{verbatim}
\]
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:

```powershell
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
```

This code for a CommandButton saves the document in the RichTextEdit `rte_1` in a text file with UTF-16LE encoding:

```powershell
integer li_rtn
li_rtn = rte_1.SaveDocument("c:\test.txt", &
    FileTypeText!, EncodingUTF16LE!)
```

### See also

- `InsertDocument`

---

**SaveInk**

**Description**

Saves ink to a file or blob from an InkPicture control.

**Applies to**

InkPicture controls

**Syntax**

```powershell
inkpicname.SaveInk ( t | b {, format {, mode } } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inkpicname</code></td>
<td>The name of the InkPicture control from which you want to save ink.</td>
</tr>
<tr>
<td><code>t</code></td>
<td>A string containing the name and location of a file that will hold the ink you want to save from the control.</td>
</tr>
<tr>
<td><code>b</code></td>
<td>The name of a blob passed by reference that will hold the ink you want to save from the control.</td>
</tr>
</tbody>
</table>
### Return value

Integer. Returns 1 for success and -1 for failure.

### Usage

Use the `SaveInk` function to save annotations made to an image in an InkPicture control to a separate file or blob.

InkSerializedFormat! (ISF) provides the most compact persistent ink representation. This format can be embedded inside a binary document format or added to the clipboard. Base64InkSerializedFormat! encodes the ISF format as a base64 stream, which allows the ink to be encoded in an XML or HTML file.

GIFFormat! saves the image in a Graphics Interchange Format (GIF) file in which ISF is embedded as metadata. This format can be viewed in applications that are not ink enabled. Base64GIFFormat! is persisted by using a base64 encoded fortified GIF. Use this format if the ink is to be encoded directly in an XML or XHTML file and will be converted to an image at a later time. It supports XSLT transformations to HTML.

### Examples

The following example saves the ink in an InkPicture control into an ISF file with default compression:

```powerscript
int li_return
string ls_pathname, ls_filename

GetFileSaveName("Save As", ls_pathname, ls_filename, "ISF")
li_return = ip_1.SaveInk(ls_pathname)
```

<table>
<thead>
<tr>
<th><strong>Argument</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
</table>
| `format` (optional) | A value of the InkPersistenceFormat enumerated variable that specifies the format in which you want to save the ink. Values are:  
  - Base64GIFFormat!  
  - Base64InkSerializedFormat!  
  - GIFFormat!  
  - InkSerializedFormat! (default) |
| `mode` (optional) | A value of the InkCompressionMode enumerated variable that specifies the compression mode in which you want to save the ink. Values are:  
  - DefaultCompression! (default)  
  - MaximumCompression!  
  - NoCompression! |
The following example saves the ink in an InkPicture control into a GIF file with maximum compression:

```powerbuilder
int li_return
string ls_pathname, ls_filename

GetFileSaveName("Save As", ls_pathname, ls_filename, "GIF")
li_return = ip_1.SaveInk(ls_pathname, GIFFormat!, MaximumCompression!)
```

See also
LoadInk
LoadPicture
ResetInk
ResetPicture
Save

---

**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 scrolls 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

Description

Scrolls to the next page of the document in a RichTextEdit control or RichTextEdit DataWindow.

For DataWindow syntax, see the ScrollNextPage method for DataWindows in the DataWindow Reference or the online Help.

Applies to

RichTextEdit controls

Syntax

rtename.ScrollNextPage ( )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
ScrollNextRow

ScrollNextPage and ScrollPriorPage work in the RichTextEdit control edit mode only when the HeaderFooter property of a rich text control is selected. They work in print preview mode regardless of the HeaderFooter property setting and they work for the RichText DataWindow control in edit mode whether or not the DataWindow has header or footer bands.

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

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.

For syntax specific to DataWindow controls and child DataWindows, see the ScrollNextRow method for DataWindows in the DataWindow Reference or the online Help.

Applies to

DataWindow and RichTextEdit controls

Syntax

$rtname$.ScrollNextRow ( )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rtname$</td>
<td>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.</td>
</tr>
</tbody>
</table>

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
Description
Scrolls to the prior page of the document in a RichTextEdit control or RichTextEdit DataWindow.

For syntax specific to DataWindow controls and child DataWindows, see the ScrollPriorPage method for DataWindows in the DataWindow Reference or the online Help.

Applies to
DataWindow and RichTextEdit controls

Syntax
   rttename.ScrollPriorPage ( )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rttename</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
ScrollPriorRow

ScrollNextPage and ScrollPriorPage work in the RichTextEdit control edit mode only when the HeaderFooter property of a rich text control is selected. They work in print preview mode regardless of the HeaderFooter property setting and they work for the RichText DataWindow control in edit mode whether or not the DataWindow has header or footer bands.

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:

```powershell
rte_1.ScrollPriorPage()
```

See also

Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorRow

ScrollPriorRow

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.

For syntax specific to DataWindow controls and child DataWindows, see the `ScrollPriorRow` method for DataWindows in the `DataWindow Reference` or the online Help.

Applies to

DataWindow and RichTextEdit controls

Syntax

```powershell
rtename.ScrollPriorRow()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

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

Description
Scrolls to the document instance associated with the specified row when the RichTextEdit controls shares data with a DataWindow.

For syntax specific to DataWindow controls and child DataWindows, see the ScrollToRow method for DataWindows in the DataWindow Reference or the online Help.

Applies to
RichTextEdit controls

Syntax
rtename.ScrollToRow ( row )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control in which you want to scroll to a document instance associated with the specified row.</td>
</tr>
<tr>
<td>row</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

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:

```plaintext
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>The time value from which you want the seconds</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns the seconds portion of time (00 to 59). If time is null, Second returns null.

Examples
This statement returns 31:

```plaintext
Second(19:01:31)
```

See also
Hour
Minute
Second method for DataWindows in the DataWindow Reference or the online Help
SecondsAfter

Description
Determines the number of seconds one time occurs after another.

Syntax
SecondsAfter ( time1, time2 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time1</td>
<td>A time value that is the start time of the interval being measured</td>
</tr>
<tr>
<td>time2</td>
<td>A time value that is the end time of the interval</td>
</tr>
</tbody>
</table>

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 method for DataWindows in the DataWindow Reference or the online Help
Seek

Moves the file pointer in an OLE stream object or displays a specified frame in an AVI clip in an animation control.

<table>
<thead>
<tr>
<th>To</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move the read/write pointer in an OLE stream object.</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Displays a specific frame in an AVI clip</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For OLE stream objects**

Moves the read/write pointer to the specified position in an OLE stream object. The pointer is the position in the stream at which the next read or write begins.

**Applies to**

OLEStream objects

**Syntax**

`olestream.Seek ( position {, origin } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olestream</code></td>
<td>The name of an OLE stream variable that has been opened.</td>
</tr>
<tr>
<td><code>position</code></td>
<td>A long whose value is the position relative to <code>origin</code> to which you want to move the read/write pointer.</td>
</tr>
<tr>
<td><code>origin</code> (optional)</td>
<td>The value of the SeekType enumerated datatype specifying where you want to start the seek. Values are:</td>
</tr>
<tr>
<td></td>
<td>• FromBeginning! – (Default) At the beginning of the file</td>
</tr>
<tr>
<td></td>
<td>• FromCurrent! – At the current position</td>
</tr>
<tr>
<td></td>
<td>• FromEnd! – At the end of the file</td>
</tr>
</tbody>
</table>

**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 does not check the functions’ return values for success, but you should be sure to check the return values in your code:

```plaintext
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

**Syntax 2**

**For animation controls**

**Description**
Displays a specific frame in an AVI clip in an animation control.

**Applies to**
Animation controls

**Syntax**

```
animationname.Seek ( s )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animationname</td>
<td>The name of animation control displaying the AVI clip</td>
</tr>
<tr>
<td>s</td>
<td>A long value in the range 0 to 65,535 indicating the frame to display</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 1 for success and -1 for failure.

**Usage**
Seek displays the specified frame. If you specify a value that is greater than the number of frames in the clip, Seek displays the last frame in the clip and returns 1. If you specify a value that is not in the specified range, Seek does nothing and returns -1. If the animation was playing, Seek always triggers the Stop event.

**Examples**
This code in a button’s clicked event displays the frame specified by a number in a single line edit control, then increments the number by one. Each click of the button advances the clip by one frame:

```plaintext
// instance variable number
integer li_return
```
SelectedColumn

number = long (sle_seek.text)
li_return = am_1.Seek(number)
number +=1
sle_seek.text = string(number)

See also
Play
Stop

**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()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</code></td>
<td>The name of the RichTextEdit in which you want the number of the character after the insertion point</td>
</tr>
</tbody>
</table>

**Return value**
Long. Returns the number of the character just after 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 `ll_col` to 5 and `ll_line` to 8:

```powershell
long ll_col, ll_line
ll_col = rte_Contact.SelectedColumn()
ll_line = rte_Contact.SelectedLine()
```

**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

```plaintext
listcontrolname.SelectedIndex ()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listcontrolname</td>
<td>The name of the ListBox or ListView control in which you want to locate the selected item</td>
</tr>
</tbody>
</table>

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:

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

These statements open the window `w_emp` if item 5 in `lb_actions` is selected:

```powerscript
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
SelectedLength

Syntax

```
listboxname.SelectedItem( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>The name of the ListBox or PictureListBox in which you want the text of the currently selected item</td>
</tr>
</tbody>
</table>

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, InkEdit, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, and DropDownListPictureListBox controls

Syntax

```
editname.SelectedItemLength( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editname</td>
<td>The name of the DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, or DropDownListPictureListBox 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.</td>
</tr>
</tbody>
</table>
### Return value
Integer for DataWindow, InkEdit, and list boxes, Long for other controls.

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
Except for text in rich text controls, the characters that make up a line ending (produced by typing Ctrl+Enter or Enter) can be 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.

---

**RichTextEdit controls**

For rich text controls, a carriage return plus a line feed always count as a single character when calculating the text length.

---

**Focus and the selection in a drop-down list**

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:

```power_script
long 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. The insertion point moves to the next line if the current line contains a carriage return.

Applies to
DataWindow, MultiLineEdit, and RichTextEdit controls

Syntax
editname.SelectLine()

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 compiles 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.SelectLine()

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.SelectLine()
    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

\[ rtename.\text{SelectedPage}() \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control in which you want the number of the current page</td>
</tr>
</tbody>
</table>

Return value Integer. Returns the number of the current page in \textit{rtename}. If an error occurs, \textit{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, \textit{SelectedPage} returns the page number within the document instance for the current row.

For more information about document instances, see \textit{DataSource}.

Examples This example returns the page number of the current page:

\begin{verbatim}
  integer li_pagect
  li_pagect = rte_1.\text{SelectedPage}()
\end{verbatim}

See also \textit{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()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>editname</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 drop-down list**
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*):

```powerbuilder
integer li_Start
li_Start = mle_Comment.SelectedStart()
```

**See also**
Position
SelectedLine
SelectedPage
SelectedText

Description
Obtains the selected text in an editable control.

Applies to
DataWindow, EditMask, InkEdit, 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.

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.

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.

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.

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.

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:

```powershell
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`:

```plaintext
string ls_Loc
ls_Loc = ddb_Location.SelectedText()
IF ls_Loc = "Maine" THEN
   lb_LBMaine.Show()
ELSE
   ... 
END IF
```

See also

`SelectText`

---

### SelectionRange

**Description**
Highlights a range of contiguous values in a trackbar control. The range you select is highlighted in the trackbar channel, with an arrow at each end of the range.

**Applies to**
Trackbar controls

**Syntax**

```plaintext
control.SelectionRange( startpos, endpos )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>control</code></td>
<td>The name of the trackbar control</td>
</tr>
<tr>
<td><code>startpos</code></td>
<td>An integer that specifies the starting position of the range</td>
</tr>
<tr>
<td><code>endpos</code></td>
<td>An integer that specifies the ending position of the range</td>
</tr>
</tbody>
</table>

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

**Usage**
Use this function to indicate a range of preferred values.

In a scheduling application, the selection range could indicate a block of time that is unavailable. Setting a selection range does not prevent the user from selecting a value either inside or outside the range.

**Examples**
This statement highlights the trackbar values between 30 and 70:

```plaintext
HTrackBar.SelectionRange( 30, 70 )
```

See also

HTrackBar in PowerBuilder *Objects and Controls*
VTrackBar in PowerBuilder *Objects and Controls*
**SelectItem**

Finds and highlights an item in a ListBox, DropDownListBox, or TreeView control.

<table>
<thead>
<tr>
<th>To select an item</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>In a ListBox control when you know the text of the item, but not its position</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>In a ListBox control when you know the position of the item in the control’s list, or to clear the current selection</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>In a TreeView control</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

**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**

\[
\text{listboxname.SelectItem ( item, index )}
\]

**Argument** | **Description**                                                                
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>The name of the ListBox control in which you want to select a line</td>
</tr>
<tr>
<td>item</td>
<td>A string whose value is the starting text of the item you want to select</td>
</tr>
<tr>
<td>index</td>
<td>The number of the item after which you want to begin the search</td>
</tr>
</tbody>
</table>

**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 drop-down list
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:

```powerbuilder
ingterger 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:

```powerbuilder
ingterger 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

```powerbuilder
listboxname.SelectItem (itemnumber)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>The name of the ListBox control in which you want to select an item</td>
</tr>
<tr>
<td>itemnumber</td>
<td>An integer whose value is the location (index) of the item in the ListBox or the ListBox portion of the drop-down list. 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 text box.</td>
</tr>
</tbody>
</table>
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 text box of a drop-down list  
To clear the text box of a DropDownListBox or DropDownPictureListBox that the user cannot edit, set itemnumber to 0. Setting the control’s text to the empty string does not 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 text box of the DropDownListBox ddb_choices and sets li_Index to 0:

```
integer li_Index
li_Index = ddb_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 )`
SelectObject

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The name of the TreeView control in which you want to select an item</td>
</tr>
<tr>
<td>itemhandle</td>
<td>The handle of the specified item</td>
</tr>
</tbody>
</table>

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:

```c
long ll_tv, ll_tvparent
int li_tvret
ll_tv = tv_list.FindItem(CurrentTreeItem, 0)
ll_tvparent = tv_list.FindItem(ParentTreeItem, &
   ll_tv)
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.

Applies to: OLE controls

Syntax: `olecontrol.SelectObject (selectstate)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olecontrol</td>
<td>The name of the OLE control containing the object you want to select</td>
</tr>
<tr>
<td>selectstate</td>
<td>A boolean value indicating whether you want to select or deselect the object</td>
</tr>
</tbody>
</table>

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:

```powerscript
integer result
result = ole_1.SelectObject(TRUE)
```

**SelectTab**

Description Selects the specified tab, displaying its tab page in the Tab control.

Applies to Tab controls

Syntax
```
tabcontrolname.SelectTab (tabidentifier)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabcontrolname</td>
<td>The name of the Tab control in which you want to select a tab</td>
</tr>
<tr>
<td>tabidentifier</td>
<td>The tab you want to select. You can specify:</td>
</tr>
<tr>
<td></td>
<td>• The tab page index (an integer)</td>
</tr>
<tr>
<td></td>
<td>• The name of the user object (datatype DragObject or UserObject)</td>
</tr>
<tr>
<td></td>
<td>• A string holding the name of the user object</td>
</tr>
</tbody>
</table>

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:

```powerscript
tab_1.SelectedTab = 3
```

Examples These three examples 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:

```powerscript
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:

```powerscript
userobject uo_tabpage
string ls_tabpage
```
SelectText

Selects text in an editable control.

### 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, InkEdit, MultiLineEdit, SingleLineEdit, DropDownListBox, and DropDownPictureListBox controls

**Syntax**

```powerbuilder
editname.SelectText ( start, length )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>editname</code></td>
<td>The name of the DataWindow, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, DropDownListBox, or DropDownPictureListBox control in which you want to select text.</td>
</tr>
<tr>
<td><code>start</code></td>
<td>A long specifying the position at which you want to start the selection.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>A long specifying the number of characters you want to select. If <code>length</code> is 0, no text is selected but PowerBuilder moves the insertion point to the location specified in <code>start</code>.</td>
</tr>
</tbody>
</table>

**Return value**

Integer for DataWindow and list boxes, Long for other controls.

Returns the number of characters selected. If an error occurs, `SelectText` returns -1.

---

```powerbuilder
ls_tabpage = "uo_fontsettings"
tab_1.OpenTab(uo_tabpage, ls_tabpage, 0)
tab_1.SelectTab(uo_tabpage)

See also

OpenTab
```

---

Selects text in an editable control.

### 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, InkEdit, MultiLineEdit, SingleLineEdit, DropDownListBox, and DropDownPictureListBox controls

**Syntax**

```powerbuilder
editname.SelectText ( start, length )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>editname</code></td>
<td>The name of the DataWindow, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, DropDownListBox, or DropDownPictureListBox control in which you want to select text.</td>
</tr>
<tr>
<td><code>start</code></td>
<td>A long specifying the position at which you want to start the selection.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>A long specifying the number of characters you want to select. If <code>length</code> is 0, no text is selected but PowerBuilder moves the insertion point to the location specified in <code>start</code>.</td>
</tr>
</tbody>
</table>

**Return value**

Integer for DataWindow and list boxes, Long for other controls.

Returns the number of characters selected. If an error occurs, `SelectText` returns -1.
CHAPTER 10  PowerScript Functions

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.

Examples

To select text in a DataWindow with the RichTextEdit presentation style, use Syntax 2.

This statement sets the insertion point at the end of the text in the SingleLineEdit sle_name:

\[
\text{sle_name.SelectText(Len(sle_name.Text), 0)}
\]

This statement selects the entire contents of the SingleLineEdit sle_name:

\[
\text{sle_name.SelectText(1, Len(sle_name.Text))}
\]

The rest of these examples assume the MultiLineEdit mle_EmpAddress contains Boston Street.

The following statement selects the string ost and returns 3:

\[
\text{mle_EmpAddress.SelectText(2, 3)}
\]

The next statement selects the string oston Street and returns 12:

\[
\begin{align*}
\text{mle_EmpAddress.SelectText(2, &} \\
&\text{Len(mle_EmpAddress.Text))}
\end{align*}
\]

These statements select the string Bos, returns 3, and sets the focus to mle_EmpAddress so that Bos is highlighted:

\[
\begin{align*}
\text{mle_EmpAddress.SelectText(1, 3)} \\
\text{mle_EmpAddress.SetFocus()}
\end{align*}
\]

See also

Len
Position
SelectedItem
SelectedText
SetFocus
TextLine
SelectText

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 } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>fromline</code></td>
<td>A long specifying the line number where the selection starts.</td>
</tr>
<tr>
<td><code>fromchar</code></td>
<td>A long specifying the number in the line of the first character in the selection.</td>
</tr>
<tr>
<td><code>toline</code></td>
<td>A long specifying the line number where the selection ends. To specify an insertion point, set <code>toline</code> and <code>tochar</code> to 0.</td>
</tr>
<tr>
<td><code>tochar</code></td>
<td>A long specifying the number in the line of the character before which the selection ends.</td>
</tr>
<tr>
<td><code>band</code> (optional)</td>
<td>A value of the Band enumerated datatype specifying the band in which to make the selection. Values are:</td>
</tr>
<tr>
<td></td>
<td>• Detail!</td>
</tr>
<tr>
<td></td>
<td>• Header!</td>
</tr>
<tr>
<td></td>
<td>• Footer!</td>
</tr>
<tr>
<td></td>
<td>The default is the band that contains the insertion point.</td>
</tr>
</tbody>
</table>

Return value
Long. Returns the number of characters selected. A carriage return with a line feed counts as a single character. If an error occurs `SelectText` returns -1. If any argument’s value is null, it 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:

```power_script
rte_1.SelectText(1,1,3,4)
```

This statement sets the insertion point at the beginning of line 2:

```power_script
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 extends past the end of the line:

```power_script
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:

```power_script
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)
```

```power_script
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 characters you can select, so the code adjusts for it:

```power_script
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
SelectTextAll

Selects all the contents of a RichTextEdit control including any special characters such as carriage return and end-of-file (EOF) markers.

Applies to RichTextEdit and DataWindow controls

Syntax

e 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 datatype 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. A carriage return with a line feed counts as a single character. If an error occurs, SelectTextAll returns -1.

Examples

This statement selects all the text in the detail band:

e rte_1.SelectTextAll()

This statement selects all the text in the header band:

e rte_1.SelectTextAll(Header!)

See also SelectedText, SelectTextAll, SelectTextLine, SelectTextWord
SelectTextLine

Description
Selects the line containing the insertion point in a RichTextEdit control.

Applies to
RichTextEdit and DataWindow controls

Syntax
rtename.SelectTextLine ( )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>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.</td>
</tr>
</tbody>
</table>

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).

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 ( )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit or DataWindow control in which you want select a word. The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow.</td>
</tr>
</tbody>
</table>
**SelectTextWord**

**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. A word can include underscores and single quotes but does not 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:

```powerbuilder
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:

```powerbuilder
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
```

---

972 PowerBuilder
// 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
\textbf{Send ( handle, message\#, lowword, long )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>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.</td>
</tr>
<tr>
<td>message#</td>
<td>An UnsignedInteger whose value is the system message number of the message you want to send.</td>
</tr>
<tr>
<td>lowword</td>
<td>A long whose value is the integer value of the message. If this argument is not used by the message, enter 0.</td>
</tr>
<tr>
<td>long</td>
<td>The long value of the message or a string.</td>
</tr>
</tbody>
</table>

Return value
Long. Returns the value returned by \texttt{SendMessage} in Windows if it succeeds and \(-1\) if an error occurs. If any argument’s value is \texttt{null}, \texttt{Send} returns \texttt{null}.

Usage
PowerBuilder’s \texttt{Send} function sends the message identified by \texttt{message\#} and optionally, \texttt{lowword} and \texttt{long}, to the window identified by \texttt{handle} to the Windows function \texttt{SendMessage}. The message is sent directly to the object, bypassing the object’s message queue. \texttt{Send} waits until the message is processed and obtains the value returned by \texttt{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:

\[
\text{Send}(\text{Handle}(w\text{-}emp), \text{277}, 2, 0)
\]

Both of the following statements click the CommandButton \`cb_OK\`:

\[
\text{Send}(\text{Handle(Parent)}, \text{273}, 0, \text{Handle(cb\text{-}OK)})
\]

\[
\text{cb\text{-}OK.TriggerEvent(Clicked!)}
\]

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:

\[
\text{Send}(\text{Handle(dw\_whatever)}, \text{274}, \text{61472}, 0)
\]

This statement maximizes the DataWindow:

\[
\text{Send}(\text{Handle(dw\_whatever)}, \text{274}, \text{61488}, 0)
\]

This statement returns the DataWindow to its normal, defined size:

\[
\text{Send}(\text{Handle(dw\_whatever)}, \text{274}, \text{61728}, 0)
\]

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:

\[
\text{// Send the Windows message for LB\_GETCURSEL}
\]
// to the list box
idx = Send(Handle(This), 1033, 0, 0)
idx = idx + 1

See also
Handle
Post

SeriesCount

Description
Counts the number of series in a graph.

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

Syntax
controlname.SeriesCount( { graphcontrol} )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph for which you want the number of series, or the name of the DataWindow control containing the graph</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>A string whose value is the name of the graph in the DataWindow control for which you want the number of series (optional)</td>
</tr>
</tbody>
</table>

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:

```powerscript
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:

```powerscript
integer li_series_count
li_series_count = &dw_equipment.SeriesCount("gr_computers")
```

See also
CategoryCount
DataCount
SeriesName

Description: Obtains the series name associated with the specified series number.

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

Syntax:  
```
controlname.SeriesName ( { graphcontrol, } seriesnumber )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want the name of a series, or the name of the DataWindow containing the graph.</td>
</tr>
<tr>
<td>graphcontrol (DataWindow control only) (optional)</td>
<td>A string whose value is the name of the graph in the DataWindow control for which you want the name of a series.</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number of the series for which you want to obtain the name.</td>
</tr>
</tbody>
</table>

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.
SetAbort

Declares that a transaction on a transaction server should be rolled back.

<table>
<thead>
<tr>
<th>To roll back a transaction</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>For OLETxnObject objects</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>For TransactionServer objects</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For OLETxnObject objects**

- **Description**: Declares that the current transaction should be rolled back.
- **Applies to**: OLETxnObject objects
- **Syntax**: `oletxnobject.SetAbort()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>oletxnobject</code></td>
<td>The name of the OLETxnObject variable that is connected to the COM object</td>
</tr>
</tbody>
</table>

- **Return value**: Integer. Returns 1 if it succeeds and -1 if an error occurs.
- **Usage**: Call the SetAbort function from the client to force a COM+ transaction to be rolled back. The default is to complete the transaction if all participants in the transaction on the COM+ server have called SetComplete or EnableCommit.
- **Examples**: The following example shows the use of SetAbort in a component method that performs database updates:

```powerscript
integer li_rc
OleTxnObject lotxn_obj

lotxn_obj = CREATE OleTxnObject
li_rc = lotxn_obj.ConnectToNewObject("pbcom.n_test")
IF li_rc <> 0 THEN
  MessageBox( "Connect Error", string(li_rc) )
  // handle error
END IF

lotxn_obj.f_dowork()
lotxn_obj.f_domorework()

IF /* test for client satisfaction */ THEN
  lotxn_obj.SetComplete()
ELSE
  lotxn_obj.SetAbort()
```
**SetAbort**

```plaintext
END IF
lotxn_obj.DisconnectObject()
```

See also: SetComplete

### Syntax 2 For TransactionServer objects

**Description**
Declares that a component cannot complete its work for the current transaction and that the transaction should be rolled back. The component instance are deactivated when the method returns.

**Applies to**
TransactionServer objects

**Syntax**
```plaintext
transactionserver.SetAbort()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transactionserver</td>
<td>Reference to the TransactionServer service instance</td>
</tr>
</tbody>
</table>

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

**Usage**
The `SetAbort` function corresponds to the `rollbackWork` transaction primitive in EAServer.

Any component that participates in a transaction can roll back the transaction by calling the `rollbackWork` primitive. Only the action of the root component (the component instance that began the transaction) determines when EAServer commits the transaction.

**Examples**
The following example shows the use of `SetAbort` in a component method that performs database updates:

```plaintext
// Instance variables:
// DataStore ids_datastore
// TransactionServer ts

Integer li_rc
long ll_rv

li_rc = this.GetContextService("TransactionServer", ts)
IF li_rc <> 1 THEN
    // handle the error
END IF
...
ll_rv = ids_datastore.Update()
IF ll_rv = 1 THEN
    ts.SetComplete()
```
SetAlignment

Description
Sets the alignment of the selected paragraphs in a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
```
rtename.SetAlignment(align)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the RichTextEdit control in which you want to set the alignment of selected paragraphs.</td>
</tr>
</tbody>
</table>
| align    | A value of the Alignment enumerated datatype 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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olename</code></td>
<td>The name of the object for which you want to set the automation locale.</td>
</tr>
<tr>
<td><code>language</code></td>
<td>A value of the LanguageID enumerated datatype 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:</td>
</tr>
<tr>
<td></td>
<td>- LanguageNeutral! – No language is assumed. Automation commands match the server’s default command set.</td>
</tr>
<tr>
<td></td>
<td>- LanguageUserDefault! – The language locale is taken from the user’s settings in the International control panel.</td>
</tr>
<tr>
<td></td>
<td>- 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!</td>
</tr>
<tr>
<td></td>
<td>- For the list of language-specific values for LanguageID, use the PowerBuilder Browser.</td>
</tr>
<tr>
<td><code>sortorder</code></td>
<td>A value of the LanguageSortID enumerated datatype specifying the sort order for the language. Values are:</td>
</tr>
<tr>
<td></td>
<td>- LanguageSortNative! – Use the traditional sort order of the selected language.</td>
</tr>
<tr>
<td></td>
<td>- LanguageSortUnicode! – Use the sort order defined for Unicode</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage

For most situations, you do not need to call SetAutomationLocale. If an automation command fails, PowerBuilder makes 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!)
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>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.</td>
</tr>
<tr>
<td>object</td>
<td>The name of an OLEObject variable that contains the automation pointer you want to use to set the pointer value in oleobject.</td>
</tr>
</tbody>
</table>

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 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!)
```
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 descendant object, saves it to a different file, and destroys both objects:

```powerbuilder
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
```

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**

```powerbuilder
oleobject.SetAutomationTimeout ( interval )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oleobject</td>
<td>The name of an OLEObject variable containing the object for which you want to set the timeout period.</td>
</tr>
<tr>
<td>interval</td>
<td>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.</td>
</tr>
</tbody>
</table>
### SetAutomationTimeout

<table>
<thead>
<tr>
<th>Return value</th>
<th>Integer. Returns 0 if it succeeds and -1 if it fails.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>This function passes the value of <em>interval</em> 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.</td>
</tr>
</tbody>
</table>

### Default timeout period

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 is too short, you may get a PowerBuilder application execution error, R0035. In this case, use SetAutomationTimeout to lengthen the timeout period.

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):

```powerbuilder
OLEObject ole1
integer rs
long interval

interval = 900000
ole1 = create OLEObject
rs = ole1.ConnectToObject("Excel.Application")
rs = ole1.SetAutomationTimeout(interval)
```
SetBoldDate

Description
Displays the specified date in bold.

Applies to
MonthCalendar control

Syntax
`controlname.SetBoldDate( d, onoff [, rt ] )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the MonthCalendar control in which you want to clear the bold dates</td>
</tr>
<tr>
<td><code>d</code></td>
<td>The date to be set in bold</td>
</tr>
<tr>
<td><code>onoff</code></td>
<td>A boolean specifying whether the date is to be set to bold. Values are:</td>
</tr>
<tr>
<td></td>
<td>true – Set the date to bold</td>
</tr>
<tr>
<td></td>
<td>false – Clear the date’s bold setting</td>
</tr>
<tr>
<td><code>rt</code> (optional)</td>
<td>A value of the MonthCalRepeatType enumerated variable. Values are:</td>
</tr>
<tr>
<td></td>
<td>Once! – Set or clear the bold setting for the specified date (default)</td>
</tr>
<tr>
<td></td>
<td>Monthly! – Using the day portion of the specified date, set or clear the bold setting for this day in all months</td>
</tr>
<tr>
<td></td>
<td>Yearly! – Using the day and month portion of the specified date, set or clear the bold setting for this date in all years</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 0 for success and one of the following negative values for failure:
-1 Invalid arguments
-2 Unknown failure

Usage
You can use the SetBoldDate function to specify that a selected date, such as an anniversary date, displays in bold. If a specific date has been set to bold, you can clear the bold setting by passing false as the second parameter.
ClearBoldDates clears all such settings.

Examples
This example sets the date January 5, 2005 to bold in the control mcVacation:

```powerscript
integer li_return
Date d
d = date("January 5, 2005")
li_return = mcVacation.SetBoldDate( d, true)
```
This example sets the fifth day of every month to bold in the control
mcVacation:

```powershell
integer li_return
Date d
d = date("January 5, 2005")
li_return = mcVacation.SetBoldDate( d, true, Monthly!)
```

This example sets the date January 5 to bold for all years in the control
mcVacation:

```powershell
integer li_return
Date d
d = date("January 5, 2005")
li_return = mcVacation.SetBoldDate( d, true, Yearly!)
```

This example clears the bold setting for the fifth day of every month in the
control mcVacation:

```powershell
integer li_return
Date d
d = date("January 5, 2005")
li_return = mcVacation.SetBoldDate( d, false, Monthly!)
```

See also

ClearBoldDates

---

### SetByte

**Description**

Sets data of type Byte for a blob variable.

**Syntax**

```
SetByte( blobvariable, n, b)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blobvariable</td>
<td>A variable of the Blob datatype in which you want to insert a value of the Byte datatype</td>
</tr>
<tr>
<td>n</td>
<td>The number of the position in blobvariable at which you want to insert a value of the Byte datatype</td>
</tr>
<tr>
<td>b</td>
<td>Data of the Byte datatype that you want to set into blobvariable at position n.</td>
</tr>
</tbody>
</table>

**Return value**

Integer. Returns 1 if it succeeds or -1 if n exceeds the scope of blobvariable; it returns null if the value of any of its arguments is null.
Examples

This example adds the byte equivalent of 37 at the initial position of the emp_photo blob. If no byte is assigned to the second position, the blob displays as the ASCII equivalent of 37 (the percent character, %) in the second message box:

```powerscript
blob {100} emp_photo
byte b1 = byte (37)
int li_rtn
li_rtn = SetByte(emp_photo, 1, b1)
messagebox("setbyte", string(b1))
messagebox("setbyte", string(emp_photo))
```

See also

Byte
GetByte

SetColumn

Description

Sets column information for a DataWindow, child DataWindow, or ListView control.

For syntax for a DataWindow or child DataWindow, see the SetColumn method for DataWindows in the DataWindow Reference or the online Help.

Applies to

ListView controls

Syntax

```
listviewname.SetColumn( index, label, alignment, width )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control for which you want to set column properties.</td>
</tr>
<tr>
<td>index</td>
<td>The number of the column for which you want to set column properties.</td>
</tr>
<tr>
<td>label</td>
<td>The label of the column for which you want to set column properties.</td>
</tr>
</tbody>
</table>
| alignment  | A value of the Alignment enumerated datatype 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. |
**SetComplete**

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

Usage: SetColumn is used only in report views.

Examples: This example sets the second column of a ListView:

```plaintext
lv_list.SetColumn(2, "Order", Center!, 800)
```

See also: AddColumn, AddItem, SetItem

---

**SetComplete**

Declares that a transaction on a transaction server should be committed.

<table>
<thead>
<tr>
<th>To commit a transaction</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>For OLETxnObject objects</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>For TransactionServer objects</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For OLETxnObject objects**

Description: Declares that the current transaction should be committed.

Applies to: OLETxnObject objects

Syntax:

```plaintext
oletxnobject.SetComplete()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oletxnobject</td>
<td>The name of the OLETxnObject variable that is connected to the COM object</td>
</tr>
</tbody>
</table>

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

Usage: Call the SetComplete function from a client to allow a COM+ transaction to be completed if all participants in the transaction on the COM+ server have called SetComplete or EnableCommit. If any participant in the transaction has called DisableCommit or SetAbort, the transaction is not completed.
The following example shows the use of `SetComplete` in a component method that performs database updates:

```power_script
integer li_rc
OleTxnObject lotxn_obj

lotxn_obj = CREATE OleTxnObject
li_rc = lotxn_obj.ConnectToNewObject("pbcom.n_test")
IF li_rc <> 0 THEN
    MessageBox( "Connect Error", string(li_rc) )
    // handle error
END IF

lotxn_obj.f_dowork()
lotxn_obj.f_domorework()
lotxn_obj.SetComplete()
lotxn_obj.DisconnectObject()
```

**See also**

`SetAbort`

---

**Syntax 2**

**For TransactionServer objects**

**Description**
Declares that the transaction in which a component is participating should be committed and the component instance should be deactivated.

**Applies to**
TransactionServer objects

**Syntax**
```
transactionserver.SetComplete( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transactionserver</code></td>
<td>Reference to the TransactionServer service instance</td>
</tr>
</tbody>
</table>

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

**Usage**
The `SetComplete` function corresponds to the `completeWork` transaction primitive in EAServer.

Any component that participates in a transaction can roll back the transaction by calling the `rollbackWork` primitive. Only the action of the root component (the component instance that began the transaction) determines when EAServer commits the transaction.
The transaction is committed if either of the following occurs:

- The root component returns with a state of `completeWork` and no participating component has set a state of `disallowCommit`.

- The root component is deactivated due to an explicit destroy from the client and no participating component has set a state of `disallowCommit`. (A client disconnect that is not preceded by an explicit destroy request always causes a rollback.)

You can use the transaction state primitives in any component; the component does not have to be declared transactional. Calling `completeWork` or `rollbackWork` from methods causes early deactivation.

**Examples**

The following example shows the use of `SetComplete` in a component method that performs database updates:

```powerbuilder
// Instance variables:
// DataStore ids_datastore
// TransactionServer ts

Integer li_rc
long ll_rv

li_rc = this.GetContextService("TransactionServer", ts)
IF li_rc <> 1 THEN
   // handle the error
END IF
...
ll_rv = ids_datastore.Update()
IF ll_rv = 1 THEN
   ts.SetComplete()
ELSE
   ts.SetAbort()
END IF
```

**See also**

- `DisableCommit`
- `EnableCommit`
- `IsInTransaction`
- `IsTransactionAborted`
- `Lookup`
- `SetAbort`
- `Which`
SetData

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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olename</code></td>
<td>The name of the OLE or custom control associated with the OLE server to which you want to transfer data.</td>
</tr>
<tr>
<td><code>clipboardformat</code></td>
<td>The format of the data. You can specify a standard format with a value of the ClipboardFormat enumerated datatype. You can specify a nonstandard format as a string. Values for ClipboardFormat are:</td>
</tr>
<tr>
<td></td>
<td>ClipFormatBitmap!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatDIB!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatDIF!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatEnhMetafile!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatHdrop!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatLocale!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatMetafilePict!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatOEMText!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatPalette!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatPenData!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatRIFF!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatSYLK!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatText!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatTIFF!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatUnicodeText!</td>
</tr>
<tr>
<td></td>
<td>ClipFormatWave!</td>
</tr>
<tr>
<td></td>
<td>If <code>clipboardformat</code> is an empty string or a null value, <code>SetData</code> transfers the data with the format ClipFormatText!.</td>
</tr>
</tbody>
</table>

| `data` | A string or blob whose value is the data you want to transfer. |

Return value

Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage

SetData returns an error if you specify a clipboard format that the OLE server does not support. See the documentation for the OLE server to find out what formats it supports.
**SetDataDDE**

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.

**Syntax**

```
SetDataDDE ( string {}, applname, topic, item )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The data you want to send to a DDE client application</td>
</tr>
<tr>
<td>applname</td>
<td>The DDE name for the client application</td>
</tr>
<tr>
<td>topic</td>
<td>A string whose value is the basic data grouping the DDE client application referenced</td>
</tr>
<tr>
<td>item</td>
<td>A string (data within topic)</td>
</tr>
</tbody>
</table>

**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 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`

### SetDataLabelling

**Description**
Set the data label for a DirectX 3D graph.

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

**Syntax**

```
controlname.SetDataLabelling ([graphcontrol], series, datapoint, value)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph from which you want data, or the name of the DataWindow control containing the graph.</td>
</tr>
</tbody>
</table>
SetDataLabelling

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control.</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number that identifies the series in which you want to label a data point.</td>
</tr>
<tr>
<td>datapoint</td>
<td>The data point that you want to label.</td>
</tr>
<tr>
<td>value</td>
<td>Indicates whether to label the data with its value.</td>
</tr>
</tbody>
</table>

Return value

Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, SetDataLabelling returns null.

Usage

SetDataLabelling is used to indicate whether or not to label the data with the numbers for DirectX 3D Area, Bar, Col, or Line graphs. You cannot use this method with DirectX 3D Pie graphs.

Examples

These statements set the series and datapoint for the graph gr_1 in the DataWindow control dw_employee.

```powerbuilder
integer SeriesNbr, ItemNbr
grObjectType clickedtype

// Get the number of the series and datapoint
clickedtype = this.ObjectAtPointer("gr_1", & SeriesNbr, ItemNbr)

// Set data label
dw_employee.SetDataLabelling("gr_1", & SeriesNbr, ItemNbr, true)
```

These statements set the series and datapoint for the graph gr_1.

```powerbuilder
integer SeriesNbr, ItemNbr
grObjectType clickedtype

clickedtype = gr_1.ObjectAtPointer(SeriesNbr, & ItemNbr)

gr_1.SetDataLabeling(SeriesNbr, ItemNbr, true)
```

See also

GetDataLabelling
GetSeriesLabelling
SetSeriesLabelling
**SetDataPieExplode**

**Description**
Explodes a pie slice in a pie graph. The exploded slice is moved away from the center of the pie, which draws attention to the data. You can explode any number of slices of the pie.

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

**Syntax**
```powerscript
controlname.SetDataPieExplode( { graphcontrol, } seriesnumber, datapoint, percentage )
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to explode a pie slice, or the name of the DataWindow containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>A string whose value is the name of the graph in the DataWindow control in which you want to explode a pie slice. (optional)</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number that identifies the series.</td>
</tr>
<tr>
<td>datapoint</td>
<td>The number of the data point (that is, the pie slice) to be exploded.</td>
</tr>
<tr>
<td>percentage</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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:

```powerscript
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**

Specifies the colors of a data point in a graph.

**Description**

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

**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 containing the graph.
*graphcontrol* | A string whose value is the name of the graph in the DataWindow control in which you want to set the color of a data point. (DataWindow control only) (optional)
*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.
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.

Using SetDataStyle with DirectX 3D Graphs

You can only set the color for the foreground. Background, line color, and shade are not supported.

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)
```

Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colorype</td>
<td>A value of the grColorType enumerated datatype specifying the aspect of the data point for which you want to set the color. Values are:</td>
</tr>
<tr>
<td></td>
<td>• Foreground! – Text color</td>
</tr>
<tr>
<td></td>
<td>• Background! – Background color</td>
</tr>
<tr>
<td></td>
<td>• LineColor! – Line color</td>
</tr>
<tr>
<td></td>
<td>• Shade! – Shade (for graphics that are three-dimensional or have solid objects)</td>
</tr>
<tr>
<td>color</td>
<td>A long whose value is the new color for colorype.</td>
</tr>
</tbody>
</table>
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:

```powerbuilder
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

Syntax
- `controlname.SetDataStyle ( { graphcontrol, } seriesnumber, datapointnumber, linestyle, linewidth )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>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 containing the graph.</td>
</tr>
<tr>
<td><code>graphcontrol</code></td>
<td>A string whose value is the name of the graph in the DataWindow control in which you want to set the line style and width. (DataWindow control only) (optional)</td>
</tr>
<tr>
<td><code>seriesnumber</code></td>
<td>The number of the series in which you want to set the line style and width of a data point.</td>
</tr>
<tr>
<td><code>datapointnumber</code></td>
<td>The number of the data point for which you want to set the line style and width.</td>
</tr>
</tbody>
</table>
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:

```powerbuilder
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)
```

### Arguments and Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linestyle</td>
<td>A value of the LineStyle enumerated datatype. Values are: Continuous! Dash! DashDot! DashDotDot! Dot! Transparent!</td>
</tr>
<tr>
<td>linewidth</td>
<td>An integer whose value is the width of the line in pixels.</td>
</tr>
</tbody>
</table>
SetDataStyle

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

Syntax
`controlname.SetDataStyle( { graphcontrol, } seriesnumber, datapointnumber, enumvalue )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to set the appearance of a data point, or the name of the DataWindow containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>A string whose value is the name of the graph in the DataWindow control in which you want to set the appearance. (DataWindow control only) (optional)</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number of the series in which you want to set the appearance of a data point.</td>
</tr>
<tr>
<td>datapointnumber</td>
<td>The number of the data point for which you want to set the appearance.</td>
</tr>
</tbody>
</table>
### CHAPTER 10  PowerScript Functions

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| enumvalue | An enumerated datatype 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.  

**Using SetDataStyle with DirectX 3D Graphs**  
You cannot use a fill pattern or specify specific symbols for the data point. |
SetDataTransparency

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:

```powerbuilder
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

SetDataTransparency

Description

Sets the transparency percentage of a data point in a series in a DirectX 3D graph (those with 3D rendering).

Applies to

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

Syntax

```
controlname.SetDataTransparency ( { graphcontrol, } seriesnumber, datapoint, transparency )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to set data, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control in which you want to set the transparency value of a data point.</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number that identifies the series in which you want to set data.</td>
</tr>
<tr>
<td>datapoint</td>
<td></td>
</tr>
<tr>
<td>transparency</td>
<td></td>
</tr>
</tbody>
</table>
### CHURCHILL 10 PowerScript Functions

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datapoint</td>
<td>The number of the data point for which you want to set a transparency value.</td>
</tr>
<tr>
<td>transparency</td>
<td>Integer value for percent transparency. A value of 0 means that the data point is opaque and a value of 100 means that it is completely transparent.</td>
</tr>
</tbody>
</table>

**Return value**

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, `SetDataTransparency` returns null.

**Usage**

`SetDataTransparency` sets the transparency value for a data point in any DirectX 3D graph (those with 3D rendering).

**Examples**

These statements set the transparency percentage to 50% for the clicked data point in the graph `gr_1` in the DataWindow control `dw_employee`:

```powerlanguage
   integer SeriesNbr, ItemNbr, TransNbr
   grObjectType clickedtype
   // Get the number of the series and datapoint
   clickedtype = this.ObjectAtPointer("gr_1", &
       SeriesNbr, ItemNbr)
   // The following statement sets Transparency to 50%
   TransNbr = 50
   dw_employee.SetDataTransparency("gr_1", &
       SeriesNbr, ItemNbr, TransNbr)
```

These statements set the transparency percentage to 50% for the clicked data point in the graph `gr_employee`:

```powerlanguage
   integer SeriesNbr, ItemNbr, TransNbr
   grObjectType clickedtype
   clickedtype = gr_employee.ObjectAtPointer( &
       SeriesNbr, ItemNbr)
   TransNbr = 50
   gr_employee.SetDataTransparency(SeriesNbr, &
       ItemNbr, TransNbr)
```

**See also**

FindSeries
GetDataTransparency
GetSeriesTransparency
SetSeriesTransparency
**SetDateLimits**

**Description**
Sets the maximum and minimum date limits for the calendar.

**Applies to**
MonthCalendar control

**Syntax**

```plaintext
controlname.SetDateLimits (min, max)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the MonthCalendar control for which you want to set the date limits</td>
</tr>
<tr>
<td><code>min</code></td>
<td>A date value to be set as the minimum date that can be referenced or displayed in the calendar</td>
</tr>
<tr>
<td><code>max</code></td>
<td>A date value to be set as the maximum date that can be referenced or displayed in the calendar</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 0 when both limits are set successfully and one of the following negative values otherwise:

- `-1` Invalid arguments
- `-2` Unknown failure

**Usage**
Use the `SetDateLimits` function to set minimum and maximum dates. `SetDateLimits` uses the maximum date as the minimum date and vice versa if you set a maximum date that is earlier than the minimum date.

**Examples**
This example sets the minimum and maximum dates for a control using today’s date as the minimum date and a date specified in an EditMask control as the maximum date:

```plaintext
ing integer li_return
Date mindate, maxdate

mindate = Today()
maxdate = Date(em_1.Text)
li_return = mc_1.SetDateLimits(mindate, maxdate)
```

**See also**
`GetDateLimits`

---

**SetDropHighlight**

**Description**
Highlights the specified item as the drop target.

**Applies to**
TreeView controls
SetDropHighlight

Syntax

```
treeviewname.SetDropHighlight (itemhandle)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The TreeView control in which you want to highlight an item as the target of a drag-and-drop operation</td>
</tr>
<tr>
<td>itemhandle</td>
<td>The handle of the item you want to highlight as the target in a drag-and-drop operation</td>
</tr>
</tbody>
</table>

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:

```powerscript
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 is 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)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicDescriptionArea</td>
<td>The name of the DynamicDescriptionArea, usually SQLDA.</td>
</tr>
<tr>
<td>index</td>
<td>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.</td>
</tr>
<tr>
<td>value</td>
<td>The value you want to use to fill the input parameter descriptor identified by index.</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, SetDynamicParm returns null.
**SetFirstVisible**

**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`:

```powershell
SQLDA.SetDynamicParm(1, "MA")
```

This statement fills the fourth input parameter descriptor in SQLDA with the number `01742`:

```powershell
SQLDA.SetDynamicParm(4, "01742")
```

This statement fills the third input parameter descriptor in SQLDA with the date `12-31-2002`:

```powershell
SQLDA.SetDynamicParm(3, "12-31-2002")
```

**See also**

- GetDynamicDate
- GetDynamicDateTime
- GetDynamicNumber
- GetDynamicString
- GetDynamicTime
- Using dynamic SQL
- OPEN Cursor

---

**SetFirstVisible**

**Description**

Sets the specified item as the first visible item in a TreeView control.

**Applies to**

TreeView controls

**Syntax**

```powershell
treeviewname.SetFirstVisible( itemhandle )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>treeviewname</code></td>
<td>The TreeView control in which you want to identify an item as the first visible item</td>
</tr>
<tr>
<td><code>itemhandle</code></td>
<td>The handle of the item you are identifying as the first visible item in the TreeView control</td>
</tr>
</tbody>
</table>

**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 allows; 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:

```power
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()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectname</code></td>
<td>The name of the object or control in which you want to set the focus</td>
</tr>
</tbody>
</table>

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:

```power
lb_Actions.SetFocus()
```
SetGlobalProperty

See also
SetItem
SetState
SetTop

SetGlobalProperty

Description
Sets the value of an SSL global property.

Applies to
SSLServiceProvider object

Syntax
sslServiceProvider.SetGlobalProperty (property, value)

Return value
Long. Returns one of the following values:
0 Success
-1 Unknown property
-2 Property is read only
-3 Invalid value for property
-10 An EAServer or SSL failure has occurred
-11 Bad argument list

Usage
The SetGlobalProperty function allows PowerBuilder clients that connect to
EAServer through SSL to set global SSL properties.

Any properties set using the SSLServiceProvider interface are global to all
connections made by the client to all EAServer servers. You can override any
of the global settings at the connection level by specifying them as options to
the Connection object or JaguarORB object.

Only clients can get and set SSL properties. Server components do not have
permission to use the SSLServiceProvider service.

Examples
The following example shows the use of the SetGlobalProperty function to set
the value of the cacheSize property to 300:

SSLServiceProvider ssl
long rc
this.GetContextService("SSLServiceProvider", ssl)
rc = ssl.SetGlobalProperty("cacheSize", "300")

See also
GetGlobalProperty

SetItem

Sets the value of an item in a list.

For use with DataWindows and DataStores, see the SetItem method for DataWindows in the DataWindow Reference or the online Help.

<table>
<thead>
<tr>
<th>To set the values of</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ListView control item</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>A ListView control item and column</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>A TreeView control item</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For ListView controls**

**Description**
Sets data associated with a ListView item to the property values you specify in a ListViewItem variable.

**Applies to**
ListView controls

**Syntax**

```
listviewname.SetItem ( index, { column }, item )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The ListView for which you are setting item properties</td>
</tr>
<tr>
<td>index</td>
<td>The index number of the item for which you are setting properties</td>
</tr>
<tr>
<td>column</td>
<td>The index number of the column of the item for which you want to set properties</td>
</tr>
<tr>
<td>item</td>
<td>The ListViewItem variable containing property values you want to assign to a ListView item</td>
</tr>
</tbody>
</table>

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

**Usage**
You can set properties for any ListView item with this syntax. If you do not specify a column, SetItem sets properties for the first column of an item. Only report views display multiple columns.
To add items to a ListView control, use the AddItem function. To add columns to a ListView control, use AddColumn. To set display values for the columns of a ListView item, use Syntax 2.

If you want to set column properties, such as alignment or width, use SetColumn. These column properties are independent of the ListViewItem objects.

To change pictures and other property values associated with a ListView item, use GetItem, change the property values, and use SetItem to apply the changes back to the ListView.

**Examples**

This example uses SetItem to change the state picture index for the selected lv_list ListView item:

```powerbuilder
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**

AddColumn
AddItem
GetItem
SetColumn

---

### Syntax 2

**For ListView controls**

Sets the value displayed for a particular column of a ListView item.

**Applies to**

ListView control

**Syntax**

```powerbuilder
listviewname.SetItem ( index, column, label )
```

**Argument** | **Description**
---|---
`listviewname` | The ListView control for which you are setting a display value
`index` | The index number of the item for which you are setting a display value
`column` | The index number of the column for which you want to set a display value
`label` | The string value or variable which you are assigning to the specified column of the specified ListView item

**Return value** | Integer. Returns 1 if it succeeds and -1 if an error occurs.
### Usage
You must include the column number as an argument, even if you are only assigning values to a single-column ListView control. To specify the properties for a ListView item, use Syntax 1.

### Examples
This example assigns display values to three columns in a report view for three `lv_list` ListView items:

```power_script
listviewitem l_vli
integer li_count, li_index

FOR li_index = 1 to 3
    li_count=li_count+1
    lv_list.AddItem("Category " + String(li_index), 1)
NEXT

lv_list.AddColumn("Composition", Left!, 860)
lv_list.AddColumn(" Album", Left!, 610)
lv_list.AddColumn(" 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")
```

### Syntax 3
**For TreeView controls**

**Description**
Sets the data associated with a specified item.

**Applies to**
TreeView controls

**Syntax**

```power_script
treeviewname.SetItem ( itemhandle, item )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The name of the TreeView control in which you want to set the data for a specific item</td>
</tr>
</tbody>
</table>
**SetLevelPictures**

**Description**
Sets the picture indexes for all items at a particular level.

**Applies to**
TreeView controls

**Syntax**
```c
treeviewname.SetLevelPictures(level, pictureindex, selectedpictureindex, statepictureindex, overlaypictureindex)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>treeviewname</code></td>
<td>The TreeView control in which you want to set the pictures for a given TreeView level</td>
</tr>
<tr>
<td><code>level</code></td>
<td>The TreeView level for which you are setting the picture indexes</td>
</tr>
<tr>
<td><code>pictureindex</code></td>
<td>An index from the regular picture list specifying the picture to be displayed when the item is not selected</td>
</tr>
<tr>
<td><code>selectedpictureindex</code></td>
<td>An index from the regular picture list specifying the picture to be displayed when the item is selected</td>
</tr>
</tbody>
</table>

**Argument**
- `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:

```c
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`
CHAPTER 10 PowerScript Functions

**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:

```PowerScript
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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>statepictureindex</code></td>
<td>An index from the state picture list specifying the picture to be displayed to the left of the regular picture</td>
</tr>
<tr>
<td><code>overlaypictureindex</code></td>
<td>An index from the overlay picture list specifying the picture to be displayed on top of the regular picture</td>
</tr>
</tbody>
</table>

**Description** Changes the files in the library search path of the application at runtime.

**Obsolete syntax** You can still use the old syntax with the name of the application object before the function call: `applicationname.SetLibraryList (filelist)`.

**Syntax**

```
SetLibraryList (filelist)
```
SetLibraryList

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.

To avoid problems that can occur when components share resources, you should use AddToLibraryList instead of SetLibraryList to add additional PBD files to the search list of a component deployed to EAServer.

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. You should always use GetLibraryList to return the current library search path and then append any files you want to add to this list. You can then pass the complete list in the filelist argument.

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:

    string ls_list

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filelist</td>
<td>A comma-separated list of file names. Specify the full file name with its extension. If you do not specify a path, PowerBuilder uses the system’s search path to find the file.</td>
</tr>
</tbody>
</table>
ls_list = getlibrarylist ()
CHOOSE CASE configuration
  CASE "Config1"
    SetLibraryList(ls_list + ",lib1.pbd, lib2.pbd, 
                      &
                      lib5.pbd*)
  CASE "Config2"
    SetLibraryList(ls_list + ",lib1.pbd, lib3.pbd, 
                      &
                      lib4.pbd*)
END CHOOSE

See also
AddToLibraryList
GetLibraryList

SetMask
Description: Sets the edit mask and edit mask datatype for an EditMask control.
Applies to: EditMask controls
Syntax: editmaskname.SetMask ( maskdatatype, mask )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editmaskname</td>
<td>The name of the EditMask for which you want to specify the edit mask.</td>
</tr>
<tr>
<td>maskdatatype</td>
<td>A MaskDataType enumerated datatype indicating the datatype of the mask. Values are:</td>
</tr>
<tr>
<td></td>
<td>• DateMask!</td>
</tr>
<tr>
<td></td>
<td>• DateTimeMask!</td>
</tr>
<tr>
<td></td>
<td>• DecimalMask!</td>
</tr>
<tr>
<td></td>
<td>• NumericMask!</td>
</tr>
<tr>
<td></td>
<td>• StringMask!</td>
</tr>
<tr>
<td></td>
<td>• TimeMask!</td>
</tr>
<tr>
<td>mask</td>
<td>A string whose value is the edit mask.</td>
</tr>
</tbody>
</table>

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 datatype 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 enters. 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 datatype, 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. At runtime, the locally correct symbols are displayed.

You cannot use SetMask to change the maskdatatype if the DropDownListCalendar, Spin, or UseCodeTable properties of the EditMask control are set to true. If any of these properties is set to true, set the property value to false before calling SetMask, then reset the value to true.

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:

```java
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:

```java
password_mask.SetMask(NumericMask!, "#####")
```

**SetMessage**

**Description**
Sets an error message for an object of type Throwable.

**Syntax**

```
throwableobject.SetMessage (newMessage )
```
### SetMessage

**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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>throwableobject</td>
<td>Object of type Throwable for which you want to set an error message.</td>
</tr>
<tr>
<td>newMessage</td>
<td>String containing the message you want to set. Must be surrounded by quotation marks.</td>
</tr>
</tbody>
</table>

**Return value**
None

**Usage**
Use to set a customized message on a user-defined exception object. Although it is possible to use `SetMessage` to modify the preset error messages for `RuntimeError` objects, this is not recommended.

**Examples**
This statement is an example of a message set on a user object of type `Throwable`:
```
MyException.SetMessage ("MyException thrown")
```

This example uses `SetMessage` in the try-catch block for a user-defined function that takes an input value from one text box and outputs the arccosine for that value into another text box:

```powerScript
uo_exception lu_error
Double ld_num
ld_num = Double (sle_1.text)
TRY
sle_2.text = string (acos (ld_num))
CATCH (runtimeerror er)
    lu_error = Create uo_exception
    lu_error.SetMessage("Value must be between -1" +
                        " and 1")
    Throw lu_error
END TRY
```

**See also**
GetMessage
SetNewMobiLinkPassword

Description
Sets a new password for the current MobiLink user.

Applies to
MLSynchronization, MLSync controls

Syntax
syncObject.SetNewMobiLinkPassword (newPW)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syncObject</td>
<td>The name of the synchronization object that starts a connection to</td>
</tr>
<tr>
<td></td>
<td>the synchronization server.</td>
</tr>
<tr>
<td>newPW</td>
<td>A string consisting of the new password that you want to set for MobiLink.</td>
</tr>
</tbody>
</table>

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)
SetNewMobiLinkPassword uses the properties in the synchronization object to generate a typical dbmlsync command line. The command line includes the -pi (ping) option as well as the -mn option to set the new MobiLink password \((newPW)\). The new password will not be set if any of the required properties (Datasource, Publication, MLServerVersion, or MLUser) are missing from the synchronization object.

See also
- GetCommandString
- SetParm
- SetSyncRegistryProperties
- Synchronize

SetNull

Sets a variable to null. The variable can be any datatype except for a structure or autoinstantiated object.

**Syntax**

\[
\text{SetNull} \ (\text{anyvariable})
\]

**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 datatype unless you specify a value when you declare the variable.

If you assign a value to a variable whose datatype is Any and then set the variable to null, the datatype of the null value is still the datatype of the assigned value. You cannot untype an Any variable with the SetNull function.

**Examples**

This statement sets the variable \(Salary\) to null:

\[
\text{SetNull}(Salary)
\]

**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:

```powerbuilder
// 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 datatype 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.
SetParm

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

SetParm

Description
Sets the parameters to send to the MobiLink synchronization server.

Applies to
MLSynchronization, MLSync controls

Syntax
```
SyncObject.SetParm(syncparm)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syncObject</td>
<td>The name of the synchronization object.</td>
</tr>
<tr>
<td>syncparm</td>
<td>A structure of type SyncParm containing property values that can be set as synchronization parameters.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 1 for success and -1 for failure.

Examples
The code fragment below creates an instance of an MLSync object and programmatically populates all of the necessary properties—as well as some optional properties—using an instance of the system SyncParm structure. Then it calls the Synchronize function to start the database synchronization.

```
SyncParm Parms
MLSync mySync
Long rc

mySync = CREATE MLSync
mySync.MLServerVersion = 9 // required property
mySync.Publication = 'salesapi' // required property
mySync.UseLogFile = TRUE // optional
```
mySync.LogFileName = "C:\temp\sync.log" // optional
mySync.Datasource = 'salesdb_remote' // required
Parms.MLUser = '50' // required
Parms.MLPass = 'xyz123' // required

// The following values are required if they are not
// set by the DSN
Parms.DBUser = 'dba'
Parms.DBPass = 'sql'

// Apply the property values to the sync object
mySync.SetParm(Parms)
// Launch the synchronization process
rc = mySync.Synchronize()
destroy mySync

Maintaining property settings in the MLSync object
Normally when you call SetParm(SyncParm) from an MLSync object, you
automatically override any authentication values (AuthenticateParms,
Datasource, DBUser, DBPass, EncryptionKey, MLUser, and MLPass) that you
set for properties of the MLSync object—even when the value of a particular
SyncParm property is an empty string. However, if you call SetNull to set a
particular property of the SyncParm object to NULL before you call SetParm,
the property value in the MLSync object will be used instead.

See also
GetCommandString
SetNewMobiLinkPassword
SetSyncRegistryProperties

SetPicture
Description
Assigns an image stored in a blob to be the image in a Picture control.
Applies to
Picture controls
Syntax

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>picturecontrol</td>
<td>The name of a Picture control in which you want to set the bitmap.</td>
</tr>
</tbody>
</table>
SetPointer

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bimage</td>
<td>A blob containing the new bitmap. <em>bimage</em> must be a valid picture in bitmap (BMP), Compuserve Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), run-length encoded (RLE), or Windows Metafile (WMF).</td>
</tr>
</tbody>
</table>

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 32,765 bytes at a time. To check the length of a file, call FileLength. If the file is over 32,765 bytes, 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:

```powerbuilder
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

```powerbuilder
SetPointer ( type )
```
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:

```powerbuilder
SetPointer(HourGlass!)
```

This example saves the old pointer and restores it when a long activity is completed:

```powerbuilder
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.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Move an object in a DataWindow to another band or to specify its front-to-back position within a band</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

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 )` 

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectname</code></td>
<td>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 always displays on top. <code>Objectname</code> cannot be a child window or a sheet.</td>
</tr>
</tbody>
</table>
| `position` | A SetPosType enumerated datatype. 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, SetPosition returns null.
CHAPTER 10  PowerScript Functions

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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwcontrol</td>
<td>The name of the DataWindow control or DataStore containing the object.</td>
</tr>
<tr>
<td>objectname</td>
<td>The name of the object within the DataWindow that you want to move. You assign names to the DataWindow objects in the DataWindow painter.</td>
</tr>
<tr>
<td>band</td>
<td>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 (&quot;&quot;) if you do not want to change the band</td>
</tr>
<tr>
<td>bringtofront</td>
<td></td>
</tr>
</tbody>
</table>

PowerScript Reference  1027
**SetProfileString**

**Description**
Writes a value in a profile file for a PowerBuilder application.

**Syntax**

```
SetProfileString ( filename, section, key, value )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filename</code></td>
<td>A string whose value is the name of the profile file. If you do not include the full path in <code>filename</code>, PowerBuilder searches the DOS path for <code>filename</code>.</td>
</tr>
<tr>
<td><code>section</code></td>
<td>A string whose value is the name of a group of related values in the profile file. If <code>section</code> does not exist in the file, PowerBuilder adds it.</td>
</tr>
<tr>
<td><code>key</code></td>
<td>A string whose value is the key in <code>section</code> for which you want to specify a value. If <code>key</code> does not exist in <code>section</code>, PowerBuilder adds it.</td>
</tr>
<tr>
<td><code>value</code></td>
<td>A string whose value is the value you want to specify for <code>key</code>.</td>
</tr>
</tbody>
</table>

**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.

---

**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

**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)
```
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 used in the keys or in the section label (if you use a section name such as Section=1):

```
[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 at runtime.

ProfileInt, ProfileString, and SetProfileString can read or write to files with ANSI or UTF16-LE encoding on Windows systems, and ANSI or UTF16-BE encoding on UNIX systems.

**Accessing the profile file**  SetProfileString uses 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 profile calls have time to complete before you try to read from the profile file.

**Windows registry**  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.
**SetRange**

Description

Sets a duration for a progress bar control or sets the start and end position for a trackbar control.

Applies to

Progress bar controls

Syntax

```plaintext
controlname.SetRange( startpos, endpos )
```

Return value

Integer. Returns 1 if it succeeds and -1 if there is an error.

Usage

The default range for the progress bar controls is 0 to 100.

Examples

This statement sets a range of 1 to 10 for a progress bar control:

```plaintext
HProgressBar.SetRange( 1, 10 )
```

See also

OffsetPos
SelectionRange
StepIt

---

**SetRecordSet**

Description

Sets an ADOResultSet object to obtain its data and metadata from a passed ADO Recordset.

Applies to

ADOResultSet objects

Syntax

```plaintext
adoreset.SetRecordSet( adorecordsetobject )
```

---

Examples

This statement sets the keyword `Title` in section `Position` of file `C:\PROFILE.INI` to the string `MGR`:

```plaintext
SetProfileString("C:\PROFILE.INI", &
 "Position", "Title", "MGR")
```

See also

ProfileInt
ProfileString
Return value

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

Usage

Use the SetRecordSet function to populate an ADOResultSet object with data passed in an OLEObject that contains an ADO Recordset. Record sets are returned from COM+ components as ADO Recordsets.

Examples

The following example connects to a COM+ component and calls a method on the component that returns an ADO Recordset to an OLEObject object. Then it creates an ADOResultSet object and populates it with data from the OLEObject using SetRecordSet:

```plaintext
OLEObject loo_mycomponent
OLEObject loo_ADOrecordset
ADOResultset lrs_ADOresultset
integer li_rc

loo_mycomponent = CREATE OLEObject
li_rc = loo_mycomponent.ConnectToNewObject("PB.Test")
IF li_rc <> 0 THEN
   MessageBox("Connect Failed", string(li_rc) )
   RETURN
END IF

// Use an OLEObject to hold ADO Recordset
// returned from method on COM+ component
loo_ADOrecordset = loo_mycomponent.GetTestResult()

// Create an ADOResultSet and get its data
// from OLEObject holding passed ADO Recordset
lrs_ADOresultset = CREATE ADOResultSet
lrs_ADOresultset.SetRecordSet(loo_ADOrecordset)
```

See also

CreateFrom method for DataWindows in the DataWindow Reference or the online Help
GenerateResultSet method for DataWindows in the DataWindow Reference or the online Help
GetRecordSet
SetResultSet

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adoresultset</td>
<td>An ADOResultSet object into which the function places the passed ADO Recordset.</td>
</tr>
<tr>
<td>adorecordsetobject</td>
<td>An OLEObject object that contains an ADO Recordset. Passing an OLEObject that does not contain an ADO Recordset generates an error.</td>
</tr>
</tbody>
</table>
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 )

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 fails.

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)

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 have established.
CHAPTER 10  PowerScript Functions

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.

Syntax
SetRemote (location, value, applname, topicname {, bAnsi})

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).
bAnsi (optional) | A boolean identifying whether the string to send to the DDE server is in ANSI format. If bAnsi is NULL, false, or empty, PowerBuilder will first try to send the data in a UNICODE formatted string. If bAnsi is true, PowerBuilder will try to send the data in an ANSI formatted string.

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 more information about DDE channels and warm and cold links, see the ExecRemote function.

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
SetRemote

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.

Syntax

```
SetRemote ( location, value, handle {, windowhandle } {, bAnsi})
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>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.</td>
</tr>
<tr>
<td>value</td>
<td>A string whose value you want to send to the remote application.</td>
</tr>
<tr>
<td>handle</td>
<td>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.</td>
</tr>
<tr>
<td>windowhandle</td>
<td>The handle to the window that is acting as the DDE client.</td>
</tr>
<tr>
<td>bAnsi (optional)</td>
<td>A boolean identifying whether the string to send to the DDE server is in ANSI format. If bAnsi is NULL, false, or empty, PowerBuilder will first try to send the data in a UNICODE formatted string. If bAnsi is true, PowerBuilder will try to send the data in an ANSI formatted string.</td>
</tr>
</tbody>
</table>

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 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:

```powerShell
long handle
handle = OpenChannel("Excel", "REGION.XLS")
SetRemote("R5C7", "4500", handle)
```

See also
- ExecRemote
- GetRemote
- OpenChannel

**SetResultSet**

Description Populates a new ADOResultSet object with data passed in a ResultSet object.

Applies to ADOResultSet objects

Syntax

```powerShell
adoresultset.SetResultSet ( resultSetobject )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adoresultset</td>
<td>An ADOResultSet object into which the function places the passed result set as an ADO Recordset</td>
</tr>
<tr>
<td>resultSetobject</td>
<td>A ResultSet object that contains result set data</td>
</tr>
</tbody>
</table>

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

Usage Use SetResultSet when you want to create an ADOResultSet object and populate it with data from a ResultSet object. The ResultSet object can be generated from a DataStore object using the GenerateResultSet function.

After you create the ADOResultSet object using SetResultSet, you can use the GetRecordSet function to return the ADO result set in an ADO Recordset object of type OLEObject that you can use as a native ADO Recordset object in PowerScript.

Examples See GetRecordSet.
SetSelectedDate

See also  GenerateResultSet method for DataWindows in the *DataWindow Reference* or the online Help
GetRecordSet
SetRecordSet

SetSelectedDate

Description  Selects a specified date.
Applies to  MonthCalendar control
Syntax  \( controlname.SetSelectedDate( d ) \)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the MonthCalendar control for which you want to set the selected date</td>
</tr>
<tr>
<td>( d )</td>
<td>A date value to be set as the date selected</td>
</tr>
</tbody>
</table>

Return value  Integer. Returns 0 for success and one of the following negative values otherwise:
-1  Invalid arguments
-2  Unknown failure

Usage  Use the `SetSelectedDate` function to select a single date. `SetSelectedDate` returns -1 if you try to specify a date that is outside the range of minimum and maximum dates specified with `SetDateLimits`.

`SetSelectedDate` and `SetSelectedRange` trigger a DateChanged event. You should not call either method in a DateChanged event, paticularly using the Post method.

Examples  This example sets the selected date to a date passed into a function:

```powershell
// function argument seldate
integer li_return
li_return = mc_1.SetSelectedDate(seldate)
```

See also  GetSelectedDate
SetDateLimits

1036  PowerBuilder
**SetSelectedRange**

**Description**
Sets the range of selected dates.

**Applies to**
MonthCalendar control

**Syntax**

```powerbrush
controlname.SetSelectedRange ( start, end )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the MonthCalendar control for which you want to set the selected range</td>
</tr>
<tr>
<td>start</td>
<td>A date value to be set as the earliest date selected</td>
</tr>
<tr>
<td>end</td>
<td>A date value to be set as the latest date selected</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 0 for success and one of the following negative values otherwise:

-1 Invalid arguments

-2 Unknown failure

**Usage**
Use the `SetSelectedRange` function to select a range of consecutive dates.

`SetSelectedRange` uses the start date as the end date and vice versa if you specify an end date that is earlier than the start date. You must set the `MaxSelectedCount` property to a value large enough to support the range before calling `SetSelectedRange`. `SetSelectedRange` returns -1 if the dates you specify are outside the range of minimum and maximum dates specified with `SetDateLimits`, or if the range exceeds `MaxSelectedCount`. If the start and end dates are the same, a single date is selected.

If the user scrolls the calendar with the navigation buttons when a date range is selected, the date range changes as the calendar scrolls.

`SetSelectedDate` and `SetSelectedRange` trigger a `DateChanged` event. You should not call either method in a `DateChanged` event, particularly using the `Post` method.

**Examples**
This example sets the start date of the selected range to `startdate` and the end date to `enddate`:

```powerbrush
integer li_return
Date startdate, enddate
startdate = Today()
enddate = Date("12-31-2007")
li_return = mc_1.SetSelectedRange(startdate, enddate)
```

**See also**
`GetSelectedRange`
`SetDateLimits`
SetSeriesLabelling

**Description**
Set the series label for a DirectX 3D graph.

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

**Syntax**
```
callname.SetSeriesLabelling ([graphcontrol, ] series, value)
```

**Return value**
Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, SetSeriesLabelling returns null.

**Usage**
SetSeriesLabelling is used to indicate whether or not to label the series with the data numbers for DirectX 3D Area, Bar, Col, or Line graphs. You cannot use this method with DirectX 3D Pie graphs.

**Examples**
These statements obtain the series and data point of graph gr_1 in the DataWindow control dw_employee.

```powerbuilder
integer SeriesNbr, ItemNbr
string ls_SeriesName
grObjectType clickedtype

// Get the number of the series and datapoint
clickedtype = this.ObjectAtPointer("gr_1", & SeriesNbr, ItemNbr)

// Get the name of series
ls_SeriesName = dw_employee.SeriesName("gr_1", & SeriesNbr)

// Set Series label
dw_employee.SetSeriesLabelling("gr_1", & ls_SeriesName, true)
```

These statements obtain the series and data point of graph gr_1.

```powerbuilder
integer SeriesNbr, ItemNbr
```
string ls_SeriesName
grObjectType clickedtype

clickedtype = gr_1.ObjectAtPointer(SeriesNbr, &
      ItemNbr)

ls_SeriesName = gr_1.SeriesName(SeriesNbr)

gr_1.SetSeriesLabelling(ls_SeriesName, true)

See also
GetDataLabelling
GetSeriesLabelling
SetDataLabelling

SetSeriesStyle

Specifies the appearance of a series in a graph. There are several syntaxes, depending on what settings you want to change.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set the series’ colors</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Set the line style and width</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Set the fill pattern or symbol for the series</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>Specify that the series is an overlay</td>
<td>Syntax 4</td>
</tr>
</tbody>
</table>

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

Syntax
controlname.SetSeriesStyle({ graphcontrol, } seriesname, colortype, color)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to set the color of a series, or the name of the DataWindow control containing the graph.</td>
</tr>
</tbody>
</table>
SetSeriesStyle

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>graphcontrol</code></td>
<td>A string whose value is the name of the graph in the DataWindow control for which you want to set the color of a series. (DataWindow control only) (optional)</td>
</tr>
<tr>
<td><code>seriesname</code></td>
<td>A string whose value is the name of the series for which you want to set the color.</td>
</tr>
<tr>
<td><code>colortype</code></td>
<td>A value of the grColorType enumerated datatype 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)</td>
</tr>
<tr>
<td><code>color</code></td>
<td>A long specifying the new color for <code>colortype</code>.</td>
</tr>
</tbody>
</table>

**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 do not 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.

**Using SetSeriesStyle with DirectX 3D Graphs**

You can only set the color for the foreground. Background, line color, and shade are not supported.

**Examples**

This statement sets the text (foreground) color of the series named Salary in the graph gr_emp_data to black:

```powerbuilder
ggr_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:

```powerscript
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:

```powerscript
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 objects

**Syntax**

```powerscript
controlname.SetSeriesStyle ( { graphcontrol, } seriesname, linestyle, linewidth )
```
**SetSeriesStyle**

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to set the line style and width.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>A string whose value is the name of the graph in the DataWindow control.</td>
</tr>
<tr>
<td>(optional)</td>
<td>that you want to set the line style and width.</td>
</tr>
<tr>
<td>serieasename</td>
<td>A string whose value is the name of the series for which you want to set the line style and width.</td>
</tr>
<tr>
<td>linestyle</td>
<td>A value of the LineStyle enumerated datatype. Values are:</td>
</tr>
<tr>
<td></td>
<td>Continuous!</td>
</tr>
<tr>
<td></td>
<td>Dash!</td>
</tr>
<tr>
<td></td>
<td>DashDot!</td>
</tr>
<tr>
<td></td>
<td>DashDotDot!</td>
</tr>
<tr>
<td></td>
<td>Dot!</td>
</tr>
<tr>
<td></td>
<td>Transparent!</td>
</tr>
<tr>
<td>linewidth</td>
<td>An integer specifying the width of the line in pixels.</td>
</tr>
</tbody>
</table>

**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 do not 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 line style and width for the series named Costs in the graph gr_product_data:

```powerbuilder
gr_product_data.SetSeriesStyle("Costs", &Dot!, 5)
```

**See also**  
GetDataStyle  
GetSeriesStyle  
SeriesName  
SetDataStyle
Synta 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

Syntax `controlname.SetSeriesStyle ( { graphcontrol, } seriesname, enumvalue )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the graph in which you want to set the appearance of a series, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td><code>graphcontrol</code></td>
<td>(DataWindow control only) (optional) A string whose value is the name of the graph in the DataWindow control in which you want to set the appearance.</td>
</tr>
<tr>
<td><code>seriesname</code></td>
<td>A string whose value is the name of the series in which you want to set the appearance.</td>
</tr>
<tr>
<td><code>enumvalue</code></td>
<td>A value of an enumerated datatype specifying an appearance setting for the series. Values for the FillPattern or grSymbolType enumerated datatypes follow.</td>
</tr>
</tbody>
</table>

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!
SetSeriesStyle

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 do not 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 pbn_dwngraphcreate and call SetSeriesStyle in the script for that event. The event pbn_dwngraphcreate is triggered just before a graph is created in a DataWindow object.

Using SetSeriesStyle with DirectX 3D Graphs
You cannot use a fill pattern or specify specific symbols for a series.

Examples
This statement sets the symbol used for the series named Costs in the graph gr_product_data to a plus sign:

```powershell
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:

```powershell
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

Syntax
`controlname.SetSeriesStyle (graphcontrol, } seriesname, overlaystyle )`
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 pbn_dwngraphcreate and call SetSeriesStyle in the script for that event. The event pbn_dwngraphcreate is triggered just before a graph is created in a DataWindow object.

Using SetSeriesStyle with DirectX 3D Graphs

You cannot use the overlay style for a series.

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 overlay_style = true
    then
        overlay_style = false
    end
SetSeriesTransparency

Description
Sets the transparency percentage of a series in a DirectX 3D graph (those with 3D rendering).

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

Syntax
controlname.SetSeriesTransparency( { graphcontrol, } series, transparency )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph in which you want to set a series transparency value, or the name of the DataWindow control containing the graph.</td>
</tr>
<tr>
<td>graphcontrol</td>
<td>(Optional) A string whose value is the name of the graph in the DataWindow control in which you want to set a series transparency value.</td>
</tr>
<tr>
<td>series</td>
<td>The string that identifies the series for which you want to set the transparency value.</td>
</tr>
<tr>
<td>transparency</td>
<td>Integer value for percent transparency. A value of 0 means that the series is opaque and a value of 100 means that it is completely transparent.</td>
</tr>
</tbody>
</table>

See also
GetDataStyle
GetSeriesStyle
SeriesName
SetDataStyle
### Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, `SetSeriesTransparency` returns null.

### Usage

`SetSeriesTransparency` sets the transparency value for a series in any DirectX 3D graph (those with 3D rendering).

### Examples

These statements set the transparency percentage to 50% for the clicked series in the graph `gr_1` in the DataWindow control `dw_employee`:

```power
integer SeriesNbr, ItemNbr, TransNbr
string ls_SeriesName
grObjectType clickedtype

// Get the number of the series and datapoint
clickedtype = this.ObjectAtPointer("gr_1", & SeriesNbr, ItemNbr)

// Get the name of series
ls_SeriesName = dw_employee.SeriesName("gr_1", & SeriesNbr)

// The following statement sets Transparency to 50%
TransNbr = 50

dw_employee.SetSeriesTransparency("gr_1", & ls_SeriesName, TransNbr)
```

These statements set the transparency to 50% for the clicked series in the graph `gr_employee`.

```power
integer SeriesNbr, ItemNbr, TransNbr
string ls_SeriesName
grObjectType clickedtype

clickedtype = gr_employee.ObjectAtPointer( & SeriesNbr, ItemNbr)

ls_SeriesName = gr_employee.SeriesName(SeriesNbr)

TransNbr = 50

gr_employee.SetSeriesTransparency( & ls_SeriesName, TransNbr)
```

### See also

- `FindSeries`
- `GetSeriesTransparency`
- `GetDataTransparency`
- `SetDataTransparency`
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 )`

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`:

```powerbuilder
rte_1.SetSpacing(Spacing2!)
```

This example specifies one and a half line spacing:

```powerbuilder
rte_1.SetSpacing(Spacing15!)
```

See also
SetTextColor
SetTextStyle
**SetState**

Sets the highlighted state of an item in a list box. SetState is only applicable to a list box control whose MultiSelect property is set to true.

**Applies to**  
ListBox and PictureListBox controls

**Syntax**  
`listboxname.SetState(index, state)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>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.</td>
</tr>
<tr>
<td>index</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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:

```PowerScript
lb_Actions.SetState(6, TRUE)
```

This statement deselects all items in lb_Actions:

```PowerScript
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:

```PowerScript
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
**SetSyncRegistryProperties**

**Description**
Sets synchronization properties in the local machine registry.

**Applies to**
MLSyncynchronization, MLSync controls

**Syntax**

```
SyncObject.SetSyncRegistryProperties ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyncObject</td>
<td>The name of the synchronization object.</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 1 for success and -1 for failure.

**Usage**
If necessary, the SetSyncRegistryProperties function can create a key in the Windows registry from the value of the SyncRegistryKey property of the synchronization object. Failure occurs if the SyncRegistryKey property is not set in the synchronization object.

**Examples**
The MLSync object generated by the MobiLink wizard for SQL Anywhere uses the following code in its Constructor event:

```plaintext
IF d_Registry_Builder < d_this_object_build THEN
  // First time deployment of new revision - set
  // "ObjectRevision" to this revision of the wizard
  // objects and override all registry values
  rc = this.SetSyncRegistryProperties()
ELSE
  rc = this.GetSyncRegistryProperties()
END IF
```

This code sets the values of authentication properties in the Windows registry when the build number of the running application is higher than the build number in the registry (or when the build number in the registry cannot be found).

**See also**
GetCommandString
GetSyncRegistryProperties
SetParm
SetTextColor

Description
Sets the color of selected text in a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
`rtename.SetTextColor(colornumber)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</code></td>
<td>The name of the RichTextEdit control in which you want to set the color of selected text</td>
</tr>
<tr>
<td><code>colornumber</code></td>
<td>A long specifying the color of the selected text</td>
</tr>
</tbody>
</table>

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:

```powerscript
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)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</code></td>
<td>The name of the RichTextEdit control in which you want to specify formatting for selected text.</td>
</tr>
<tr>
<td><code>bold</code></td>
<td>A boolean value specifying whether the selected text is bold.</td>
</tr>
<tr>
<td><code>underline</code></td>
<td>A boolean value specifying whether the selected text is underlined.</td>
</tr>
<tr>
<td><code>subscript</code></td>
<td>Maintained for backward compatibility only. A boolean value specifying whether the selected text is a subscript. This value is currently ignored.</td>
</tr>
<tr>
<td><code>superscript</code></td>
<td></td>
</tr>
<tr>
<td><code>italic</code></td>
<td></td>
</tr>
<tr>
<td><code>strikeout</code></td>
<td></td>
</tr>
</tbody>
</table>

PowerScript Reference 1051
**SetTimeout**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscript</td>
<td>A boolean value specifying whether the selected text is a subscript. Maintained for backward compatibility only. If both subscript and superscript are true, subscript takes precedence and the text is subscripted. This value is currently ignored.</td>
</tr>
<tr>
<td>italic</td>
<td>A boolean value specifying whether the selected text is italic.</td>
</tr>
<tr>
<td>strikeout</td>
<td>A boolean value specifying whether the selected text is has a line drawn through it.</td>
</tr>
</tbody>
</table>

**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:

```plaintext
rte_1.SetTextStyle(TRUE, FALSE, &
    TRUE, FALSE)
```

This example makes the selected text italic but keeps other text formatting as it was:

```plaintext
rte_1.SetTextStyle(rte_1.GetTextStyle(Bold!), &
    rte_1.GetTextStyle(Underlined!), &
    TRUE, rte_1.GetTextStyle(Strikeout!))
```

**See also**

GetTextStyle
SetSpacing
SetTextColor

**SetTimeout**

**Description**
Sets the timeout value for subsequent EAServer transactions. The transaction is rolled back if it does not complete before the timeout expires.

**Applies to**
CORBACurrent objects

**Syntax**

`CORBACurrent.SetTimeout ( seconds )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBACurrent</td>
<td>Reference to the CORBACurrent service instance</td>
</tr>
<tr>
<td>seconds</td>
<td>An unsignedlong that specifies the number of seconds that elapse before a transaction is rolled back</td>
</tr>
</tbody>
</table>

**Return value**

Boolean. Returns true if it succeeds and false if an error occurs.
CHAPTER 10  PowerScript Functions

Usage

The `SetTimeout` function specifies the number of seconds that can elapse before a transaction is rolled back. The timeout period applies to transactions created by subsequent invocations of `BeginTransaction`. If `seconds` is 0, no timeout period is in effect.

`SetTimeout` can be called by a client or a component that is marked as OTS style. EAServer must be using the two-phase commit transaction coordinator (OTS/XA).

Examples

This example shows the use of `SetTimeout` to set the timeout period to five minutes:

```powerScript
// Instance variables:
// CORBACurrent corbcurr
boolean lb_timeout
integer li_rc

li_rc = this.GetContextService("CORBACurrent", &
corbcurr)
IF li_rc <> 1 THEN
   // handle the error
END IF
li_rc = corbcurr.Init( "iiop://server1:9003")
IF li_rc <> 1 THEN
   // handle the error
ELSE
   lb_timeout = corbcurr.SetTimeout(300)
   li_rc = corbcurr.BeginTransaction()
END IF
```

See also

- `BeginTransaction`
- `CommitTransaction`
- `getContextService`
- `getStatus`
- `getTransactionName`
- `Init`
- `ResumeTransaction`
- `RollbackOnly`
- `RollbackTransaction`
- `SuspendTransaction`
SetToday

Description
Sets the value that is used by the calendar as today’s date.

Applies to
DatePicker, MonthCalendar controls

Syntax
controlname.SetToday( d )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the control for which you want to set the Today date</td>
</tr>
<tr>
<td>d</td>
<td>The date you want to specify as the Today date</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 0 for success and -1 for failure.

Usage
By default, the current system date is set as the Today date. You can use the
SetToday function to specify a different date. If the date is set to any date other
than the current system date, the following restrictions apply:

- The control does not automatically update the Today selection when the
time passes midnight for the current day.
- The control does not automatically update its display based on locale
changes.

Examples
This example gets a date from an EditMask control and sets it as the Today date
in a MonthCalendar control:

```
Date currentdate
integer li_return

currentdate = Date(em_1.Text)
li_return = mc_1.SetToday(currentdate)
```

See also
GetToday
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 } } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>window</td>
<td>The MDI frame or sheet to which the toolbar belongs.</td>
</tr>
<tr>
<td>toolbarindex</td>
<td>An integer whose value is the index of the toolbar whose settings you want to change.</td>
</tr>
<tr>
<td>visible</td>
<td>A boolean value specifying whether to make the toolbar visible. Values are:</td>
</tr>
<tr>
<td></td>
<td>• TRUE – Make the toolbar visible</td>
</tr>
<tr>
<td></td>
<td>• FALSE – Hide the toolbar</td>
</tr>
<tr>
<td>alignment (optional)</td>
<td>A value of the ToolbarAlignment enumerated datatype specifying the alignment for the toolbar. Values are:</td>
</tr>
<tr>
<td></td>
<td>• AlignAtTop! – Dock the toolbar at the top of the frame.</td>
</tr>
<tr>
<td></td>
<td>• AlignAtLeft! – Dock the toolbar on the left side of the frame.</td>
</tr>
<tr>
<td></td>
<td>• AlignAtRight! – Dock the toolbar on the right side of the frame.</td>
</tr>
<tr>
<td></td>
<td>• AlignAtBottom! – Dock the toolbar at the bottom of the frame.</td>
</tr>
<tr>
<td></td>
<td>• Floating! – Float the toolbar. The floating toolbar has its own frame and miniature title bar</td>
</tr>
<tr>
<td>floatingtitle (optional)</td>
<td>A string whose value is the title for the toolbar when its alignment is Floating!.</td>
</tr>
</tbody>
</table>

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.
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:

```powerbuilder
 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:

```powerbuilder
    Parent.i_toolbaralign = AlignAtTop!
```

This script changes the toolbar alignment:

```powerbuilder
    w_frame.SetToolbar(1, TRUE, &
                       uo_toolbarpos.i_toolbaralign )
```

See also

- GetToolbar
- GetToolbarPos
- SetToolbarPos
SetToolbarPos

Sets the position of the specified toolbar.

<table>
<thead>
<tr>
<th>To set</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docking position of a docked toolbar</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Coordinates and size of a floating toolbar</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

For docked toolbars

Sets the position of a docked toolbar.

**Description**

MDI frame and sheet windows

**Syntax**

`window.SetToolbarPos( toolbarindex, dockrow, offset, insert )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>window</code></td>
<td>The MDI frame or sheet to which the toolbar belongs.</td>
</tr>
<tr>
<td><code>toolbarindex</code></td>
<td>An integer whose value is the index of the toolbar whose settings you want to change.</td>
</tr>
<tr>
<td><code>dockrow</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>offset</code></td>
<td>An integer whose value specifies the distance of the toolbar from the beginning of the docking row. For toolbars at the top or bottom, <code>offset</code> is measured from the left edge. For toolbars on the left or right, <code>offset</code> is measured from the top. If <code>insert</code> is true, the <code>offset</code> you specify is adjusted so that the toolbar does not overlap others in the row. Specify an offset of 0 to position the toolbar ahead of other toolbars in <code>dockrow</code>.</td>
</tr>
<tr>
<td><code>insert</code></td>
<td>A boolean value specifying whether you want to insert the specified toolbar before the toolbars in <code>dockrow</code> causing them to move over or down a row, or you want to add <code>toolbarindex</code> to <code>dockrow</code>. Values are:</td>
</tr>
<tr>
<td></td>
<td>• TRUE – Move any toolbars already in <code>dockrow</code> or higher rows over or down a row so that the toolbar you are moving is the only toolbar in the row.</td>
</tr>
<tr>
<td></td>
<td>• FALSE – Add the toolbar you are moving to <code>dockrow</code>. Its position in relation to other toolbars in the row is determined by <code>offset</code>.</td>
</tr>
</tbody>
</table>
SetToolbarPos

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 following 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:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
w_frame.SetToolbar(1, TRUE, AlignAtLeft!)  
w_frame.SetToolbarPos(1, 2, 1, TRUE)
```

See also

GetToolbar  
GetToolbarPos  
SetToolbar

1058  
PowerBuilder
CHAPTER 10  PowerScript Functions

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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>window</code></td>
<td>The MDI frame or sheet to which the toolbar belongs</td>
</tr>
<tr>
<td><code>toolbarindex</code></td>
<td>An integer whose value is the index of the toolbar whose settings you want to change</td>
</tr>
<tr>
<td><code>x</code></td>
<td>An integer whose value is the x coordinate of the floating toolbar</td>
</tr>
<tr>
<td><code>y</code></td>
<td>An integer whose value is the y coordinate of the floating toolbar</td>
</tr>
<tr>
<td><code>width</code></td>
<td>An integer whose value is the width of the floating toolbar</td>
</tr>
<tr>
<td><code>height</code></td>
<td>An integer whose value is the height of the floating toolbar</td>
</tr>
</tbody>
</table>

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, `SetToolbarPos` 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:

```powerscript
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:

```powerbuilder
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:

```powerbuilder
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
CHAPTER 10  PowerScript Functions

SetTop
Description  Scrolls a list box control so that the specified item is the first visible item.
Applies to  ListBox and PictureListBox controls
Syntax  

```
listboxname.SetTop ( index )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>The name of the ListBox or PictureListBox that you want to scroll</td>
</tr>
<tr>
<td>index</td>
<td>The number of the item you want to become the first visible item</td>
</tr>
</tbody>
</table>

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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instancename</td>
<td>Instance name of the Profiling or TraceTree object</td>
</tr>
<tr>
<td>tracefilename</td>
<td>A string that identifies the name of the trace file PowerBuilder will analyze</td>
</tr>
</tbody>
</table>

Return value  ErrorReturn. Returns one of the following values:

- Success! – The function succeeded
- FileOpenError! – The file could not be opened
SetTraceFileName

- 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:

```power
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
CHAPTER 10  PowerScript Functions

SetTransPool
Description  Sets up a pool of database transactions for a distributed application.
SetTransPool was used with a feature that has been removed from
PowerBuilder and is an obsolete function.
Applies to  Application object
Syntax  

\texttt{applicationname.SetTransPool (minimum, maximum, timeout)}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{applicationname}</td>
<td>The name of the application object for which you want to establish a transaction pool</td>
</tr>
<tr>
<td>\texttt{minimum}</td>
<td>The minimum number of transactions to be kept open in the pool</td>
</tr>
<tr>
<td>\texttt{maximum}</td>
<td>The maximum number of transactions that can be open in the pool</td>
</tr>
<tr>
<td>\texttt{timeout}</td>
<td>The number of seconds to allow a request to wait for a connection in the transaction pool</td>
</tr>
</tbody>
</table>

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

SetValue
Description  Sets the date and time in the Value property of the control.
Applies to  DatePicker control
Syntax  

\texttt{controlname.SetValue (d, t)}
\texttt{controlname.SetValue (dt)}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{controlname}</td>
<td>The name of the control for which you want to set the date and time</td>
</tr>
<tr>
<td>\texttt{d}</td>
<td>The date value to be set in the Value property</td>
</tr>
<tr>
<td>\texttt{t}</td>
<td>The time value to be set in the Value property</td>
</tr>
<tr>
<td>\texttt{dt}</td>
<td>The DateTime value to be set in the Value property</td>
</tr>
</tbody>
</table>
### SetValue

<table>
<thead>
<tr>
<th>Return value</th>
<th>Integer. Returns 1 for success and one of the following negative values for failure:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1  The value cannot be set</td>
</tr>
<tr>
<td></td>
<td>-2  Other error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage</th>
<th>The <code>SetValue</code> function can set the Value property using separate date and time variables or a single DateTime variable.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Examples</th>
<th>This example sets the Value property of a DatePicker control using separate date and time values:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>date d</td>
</tr>
<tr>
<td></td>
<td>time t</td>
</tr>
<tr>
<td></td>
<td>d=date(&quot;2007/12/27&quot;)</td>
</tr>
<tr>
<td></td>
<td>t=time(&quot;12:00:00&quot;)</td>
</tr>
<tr>
<td></td>
<td>dp_1.SetValue(d, t)</td>
</tr>
<tr>
<td></td>
<td>This example sets the Value property using a DateTime value:</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>date d</td>
</tr>
<tr>
<td></td>
<td>time t</td>
</tr>
<tr>
<td></td>
<td>datetime dt</td>
</tr>
<tr>
<td></td>
<td>dt = DateTime(d, t)</td>
</tr>
<tr>
<td></td>
<td>dp_1.SetValue(dt)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>See also</th>
<th>GetText</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GetValue</td>
</tr>
</tbody>
</table>
**SharedObjectDirectory**

**Description**  
Retrieves the list of objects that have been registered for sharing.

**Syntax**  
`SharedObjectDirectory ( instancenames {, classnames } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instancenames</code></td>
<td>An unbounded array of type string in which you want to store the names of objects that have been registered for sharing</td>
</tr>
<tr>
<td><code>classnames</code></td>
<td>An unbounded array of type string in which you want to store the class names of objects registered for sharing (optional)</td>
</tr>
</tbody>
</table>

**Return value**  
ErrorReturn. Returns one of the following values:

- Success! – The function succeeded
- FeatureNotSupportedError! – This function is not supported on this platform

**Usage**  
Use this function to obtain a list of objects that have been registered for sharing.

**Examples**  
In this example, the application retrieves the list of shared objects and their class names:

```powerscript
errorreturn status
string InstanceNames[]
string ClassNames[]

status = SharedObjectDirectory(InstanceNames, & ClassNames)
```

**See also**  
SharedObjectGet  
SharedObjectRegister

---

**SharedObjectGet**

**Description**  
Gets a reference to a shared object instance.

**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 of type PowerObject 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  
- `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`.  
You can use a shared object on a PowerBuilder client to simulate an asynchronous call to EAServer. The main thread on the client makes an asynchronous call to a function on the shared object, passing it a callback object that is notified when processing has finished on the server. The method on the shared object makes a synchronous call to the EAServer component method that performs processing. Since the shared object is running in a separate thread on the client, the main thread on the client can proceed with other work while the process runs on the server.  

**Examples**  
This example shows how you might use a shared object to make an asynchronous request against an EAServer component method and return data back to a client application window. The client has a Retrieve button on a window, a `SetDW` function, a shared object, and a callback handler. The component deployed to EAServer retrieves employee information from a database.  
The Retrieve button on the window creates a shared object that communicates with EAServer as well as an instance of a callback handler:  

```plaintext  
// instance variables  
// uo_sharedobject iuo_sharedobject  
// uo_callback iuo_callback  
long ll_rv  
SharedObjectRegister("uo_sharedobject","myshare")  
SharedObjectGet("myshare", iuo_sharedobject)  
```
iuo_callback = CREATE uo_callback
// Pass a reference to the window to
// the callback object
iuo_callback.passobject (parent)

iuo_sharedobject.post retrievedata(iuo_callback)

The SetDW function applies the contents of the DataWindow blob returned from the EAServer component to a DataWindow control in the window:

long ll_rv
ll_rv = dw_employee.SetFullState(ablb_data)
if ll_rv = -1 then
    MessageBox("Error", "SetFullState call failed!")
end if
return ll_rv

The Constructor event of the shared object uses a custom Connection object called n_jagclnt_connect to connect to the server. Then it creates an instance of the EAServer component:

// Instance variables
// uo_employee iuo_employee
// n_jagclnt_connect myconnect
Constructor event
long ll_rc
myconnect = create n_jagclnt_connect
ll_rc = myconnect.ConnectToServer()
ll_rv = myconnect.CreateInstance(iuo_employee, & "uo_employee")

The shared object has a single function called RetrieveData that makes a synchronous call to the RetrieveData function on the EAServer component.

When the function completes processing, it calls the Notify function asynchronously on the callback object, posting it to the DataWindow blob returned from the server component:

blob lblb_data
long ll_rv
ll_rv = iuo_employee.retrievedata(lblb_data)
auo_callback.post notify(lblb_data)
return ll_rv
When the EAServer component has finished processing, the shared object
notifies a user object called `uo_callback`, which in turns notifies the `w_employee`
window. The `uo_callback` object has two functions, `Notify` and `PassObject`. The
`Notify` function calls a function called `SetDW` on the `w_employee` window,
passing it the DataWindow blob returned from the server component:

```powershell
long ll_rv
ll_rv = iw_employee.setdw(ablb_data)
if ll_rv = -1 then
    MessageBox("Error", "SetDW call failed!")
end if
return ll_rv
```

The callback handler’s `PassObject` function caches a reference to the
`w_employee` window in the `iw_employee` instance variable. The function takes
the argument `aw_employee`, which is of type `w_employee`, and returns a long
value:

```powershell
iw_employee = aw_employee
return 1
```

The EAServer component is a PowerBuilder user object called `uo_employee`.
The `uo_employee` object has a function called `RetrieveData` that uses a
DataStore to retrieve employee rows from the database:

```powershell
// instance variables
// protected TransactionServer txnsrv
// protected DataStore ids_datastore
long ll_rv
ll_rv = ids_datastore.Retrieve()
ll_rv = ids_datastore.GetFullState(ablb_data)
txnsrv.SetComplete()
return ll_rv
```

See also

- `SharedObjectRegister`
- `SharedObjectUnregister`
- `GetFullState` and `SetFullState` methods for DataWindows in the `DataWindow
Reference` or the online Help
**SharedObjectRegister**

**Description**
Registers a user object so that it can be shared.

**Syntax**
```
SharedObjectRegister ( classname, instancename )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>classname</code></td>
<td>The name of the user object that you want to share</td>
</tr>
<tr>
<td><code>instancename</code></td>
<td>A string whose value is the name you want to assign to the shared object instance</td>
</tr>
</tbody>
</table>

**Return value**
ErrorReturn. Returns one of the following values:
- Success! – The function succeeded
- 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.

You can use a shared object on a PowerBuilder client to simulate an asynchronous call to EAServer. For more information, see the description of the `SharedObjectGet` function.

**Examples**
In this example, the user object `uo_customers` is registered so that it can be shared. The name assigned to the shared object instance is `share1`. After registering the object, the application 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.

**Syntax**

```powershell
SharedObjectUnregister (instancename)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instancename</td>
<td>The name assigned to the shared object instance when it was first registered</td>
</tr>
</tbody>
</table>

**Return value**

ErrorReturn. Returns one of the following values:

- **Success!** – The function succeeded
- **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.

You can use a shared object on a PowerBuilder client to simulate an asynchronous call to EAServer. For more information, see the description of the `SharedObjectGet` function.

**Examples**

In this example the application unregisters the object instance called `share1`:

```powershell
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**

```powershell
objectname.Show()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectname</td>
<td>The name of the object or control you want to make visible (show)</td>
</tr>
</tbody>
</table>

**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 drop-down 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`:

```plaintext
objectname.Visible = true
```

This statement:

```plaintext
m_status.m_options.Visible = TRUE
```

is equivalent to:

```plaintext
m_status.m_options.Show()
```

Examples

This statement makes visible the menu selection called `m_options` on the menu `m_status`:

```plaintext
m_status.m_options.Show()
```

This statement makes the child window `w_child` visible:

```plaintext
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**

```plaintext
rtename.ShowHeadFoot ( editheadfoot, {headerfooter} )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</code></td>
<td>The name of the RichTextEdit or DataWindow control for which you want to edit header and footer information.</td>
</tr>
</tbody>
</table>
| `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 |
ShowHelp

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>helpfile</td>
<td>A boolean value specifying whether the insertion point (caret) for editing the header/footer panel is in the header or the footer section. Values are:</td>
</tr>
<tr>
<td>headerfooter</td>
<td>- True: Caret is in the header section.</td>
</tr>
<tr>
<td></td>
<td>- False: Caret is in the footer section.</td>
</tr>
</tbody>
</table>

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 headerfooter argument is ignored if the editheadfoot argument is false. The headerfooter argument defaults to “true” if a value is not provided. The header and footer can include input fields for page numbers and dates.

For a DataWindow control, ShowHeadFoot has no effect if the DataWindow object does not have the RichTextEdit presentation style.

Examples

This example displays the header and footer editing panels, allowing the user to specify the contents of the footer:

```
rte_1.ShowHeadFoot(TRUE, FALSE)
```

The following script inserts the current page number in the footer, then returns the focus to the body of the document in the rich text control:

```
rte_1.ShowHeadFoot(true,false)
rte_1.SetAlignment (Center!)
rte_1.InputFieldInsert("PAGENO")
rte_1.ShowHeadFoot(false,false)
```

See also

Preview

ShowHelp

Description

Provides access to a Microsoft Windows-based Help system or to compiled HTML Help files that you have created for your PowerBuilder application. When you call ShowHelp, PowerBuilder starts the Help executable and displays the Help file you specify.

Syntax

```
ShowHelp ( helpfile, helpcommand {, typeid} )
```

PowerBuilder
### CHAPTER 10  PowerScript Functions

**Return value**

Integer. Returns 1 if it succeeds and -1 if an error occurs. `ShowHelp` returns -1 if you specify `typeid` when `helpcommand` is Finder! or 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 information on how to create online Help files for your PowerBuilder application, see the chapter on providing online Help in *PowerBuilder Application Techniques*.

**Examples**

This statement displays the Help index in the *INQ.HLP* file:

```powerscript
ShowHelp("C:\PB\INQ.HLP", Index!)
```

This statement displays Help topic 143 in the file *EMP.HLP* file:

```powerscript
ShowHelp("EMP.HLP", Topic!, 143)
```

This statement displays the Help topic associated with the keyword Part# in the file *EMP.HLP*:

```powerscript
ShowHelp("EMP.HLP", Keyword!, "Part#")
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>helpfile</td>
<td>A string whose value is the name of the compiled HLP file or the CHM (HTML Help) file.</td>
</tr>
<tr>
<td>helpcommand</td>
<td>A value of the HelpCommand enumerated type. Values are:</td>
</tr>
<tr>
<td></td>
<td>• Finder! – Displays the Help file in its most recently used state (the Help Topics dialog box in WinHelp or the Navigator pane in the HTML Help viewer open to the last-used tab or the default tab for the Help file).</td>
</tr>
<tr>
<td></td>
<td>• Index! – Displays the top-level contents topic in the Help file.</td>
</tr>
<tr>
<td></td>
<td>• Keyword! – Goes to the topic identified by the keyword in <code>typeid</code>.</td>
</tr>
<tr>
<td></td>
<td>• Topic! – Displays the topic identified by the number in <code>typeid</code>.</td>
</tr>
<tr>
<td>typeid (optional)</td>
<td>A number identifying the topic if <code>helpcommand</code> is Topic! or a string whose value is a keyword of a help topic if <code>helpcommand</code> is Keyword!. Do not specify <code>typeid</code> when <code>helpcommand</code> is Finder! or Index!.</td>
</tr>
</tbody>
</table>
**ShowPopupHelp**

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")
```

See also  
Help  
ShowPopupHelp

**ShowPopupHelp**

**Description**  
Displays pop-up help for the specified control.

**Applies to**  
Any control

**Syntax**  

```
ShowPopupHelp ( helpfile, control, contextid )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>helpfile</td>
<td>String for the Help file name to be used</td>
</tr>
<tr>
<td>control</td>
<td>Dragobject for which the pop-up help is displayed</td>
</tr>
<tr>
<td>contextid</td>
<td>Long for the context ID number</td>
</tr>
</tbody>
</table>

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

**Usage**  
A typical location for the ShowPopupHelp call is in the Help event of a response window with the Context Help property enabled. Events relating to movement of the cursor over a control or to the dragging of a control or object are also logical places for a ShowPopupHelp call.

You must type a correct context ID number for the contextid argument or you get a message that a Help topic does not exist for the item calling the ShowPopupHelp function.

**Examples**  
This example calls a help file in a subdirectory of the current directory:

```
ShowPopupHelp ( "Help/my_app.hlp", this, 510)
```

See also  
Help  
ShowHelp
## Sign

### Description
Reports whether a number is negative, zero, or positive.

### Syntax
\[ \text{Sign} ( n ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>The number for which you want to find out the sign</td>
</tr>
</tbody>
</table>

### 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):
\[ \text{Sign}(5) \]
This statement returns 0 (zero has no sign):
\[ \text{Sign}(0) \]
This statement returns -1 (the number is negative):
\[ \text{Sign}(-5) \]

### See also
Sign method for DataWindows in the *DataWindow Reference* or online Help

## SignalError

### Description
Causes a SystemError event at the application level.

### Syntax
\[ \text{SignalError} ( \{ \text{number} \}, \{ \text{text} \} ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{number} ) (optional)</td>
<td>The integer (stored in the number property of the Error object) to be used in the message object</td>
</tr>
<tr>
<td>( \text{text} ) (optional)</td>
<td>The string (stored in the text property of the Error object) to be used in the message object</td>
</tr>
</tbody>
</table>

### 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 need to include code in the SystemError event script to recognize and handle the error you have created. If there is no script for the SystemError event, the SignalError function does nothing.

For the runtime error numbers assigned to the Number property of the Error object when an application error occurs, see the PowerBuilder Users 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:

```plaintext
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

\[
\text{Sin} \ (n)
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>The angle (in radians) for which you want the sine</td>
</tr>
</tbody>
</table>

Return value

Double. Returns the sine of \(n\). If \(n\) is null, \(\text{Sin}\) returns null.
Examples

This statement returns .8414709848078965:

\[ \text{Sin}(1) \]

This statement returns 0:

\[ \text{Sin}(0) \]

This statement returns 0:

\[ \text{Sin}(\text{Pi}(1)) \]

See also

ASin
Cos
Pi
Tan
Sin method for DataWindows in the DataWindow Reference or the online Help

---

**Sleep**

Description

Causes the application to pause for a specified time.

Syntax

\[ \text{Sleep}( \text{seconds} ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seconds</td>
<td>Long for the number of seconds you want the application to pause</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns 1 if the function succeeds and -1 if an error occurs.

Examples

This example pauses the application for 5 seconds:

\[ \text{Sleep}(5) \]

---

**Sort**

Sorts rows in a DataWindow control, DataStore, or child DataWindow, or items in a TreeView or ListView control.
For syntax for DataWindows and DataStores, see the Sort method for DataWindows in the *DataWindow Reference* or the online Help.

<table>
<thead>
<tr>
<th>To sort</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items in a TreeView</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Items in a ListView</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1  
**For TreeView controls**

**Description**
Sorts the children of an item in a TreeView control.

**Applies to**
TreeView controls

**Syntax**
```
treeviewname.Sort(itemhandle, sorttype)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>treeviewname</code></td>
<td>The name of the TreeView control in which you want to sort items.</td>
</tr>
<tr>
<td><code>itemhandle</code></td>
<td>The item for which you want to sort its children.</td>
</tr>
<tr>
<td><code>sorttype</code></td>
<td>The sort method you want to use. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>Ascending!</td>
</tr>
<tr>
<td></td>
<td>Descending!</td>
</tr>
<tr>
<td></td>
<td>UserDefinedSort!</td>
</tr>
</tbody>
</table>

**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. To sort level 1 of a TreeView, set `itemhandle` to 0.

**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 2  
**For ListView controls**

**Description**
Sorts items in ListView controls.

**Applies to**
ListView controls
Syntax

```
listviewname.Sort ( sorttype, { column } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The ListView in which you want to sort items.</td>
</tr>
<tr>
<td>sorttype</td>
<td>The method you want to use when you sort the ListView items. Values are: Ascending! Descending! Unsorted! UserDefinedSort!</td>
</tr>
<tr>
<td>column (optional)</td>
<td>The number of the column by which you wish to sort the ListView items.</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns 1 if it succeeds and -1 if it fails.

Usage

The default sort is alphanumerical.

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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>treeviewname</td>
<td>The TreeView control in which you want to sort the subsequent levels in an item’s hierarchy.</td>
</tr>
<tr>
<td>itemhandle</td>
<td>The item for which you want to sort all the levels below it.</td>
</tr>
<tr>
<td>sorttype</td>
<td>The sort method you want to use. Values are: Ascending! Descending! Unsorted! UserDefinedSort!</td>
</tr>
</tbody>
</table>
Space

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 cannot 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:

```powerbuilder
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:

```powerbuilder
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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>A long whose value is the length of the string you want filled with spaces. The maximum value is 2,147,483,647, which is the maximum size for strings.</td>
</tr>
</tbody>
</table>

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:

```plaintext
string Name
Name = Space(4)
```

This statement assigns 40 spaces to the string Name:

```plaintext
string Name
Name = Space(40)
```

See also

Fill
Space method for DataWindows in the DataWindow Reference or online Help

---

# Sqrt

## Description

Calculates the square root of a number.

## Syntax

```plaintext
Sqrt(n)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The number for which you want the square root</td>
</tr>
</tbody>
</table>

## Return value

Double. Returns the square root of \( n \). If \( n \) is null, Sqrt returns null.

## Usage

Sqrt\((n)\) is the same as \( n^{0.5} \).

Taking the square root of a negative number causes an execution error.

## Examples

This statement returns 1.414213562373095:

```plaintext
Sqrt(2)
```

This statement results in an error at execution time:

```plaintext
Sqrt(-2)
```

See also

Sqrt method for DataWindows in the DataWindow Reference or online Help
**Start**

Start has two basic syntaxes.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute a pipeline object</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Activate a timing object</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Syntax 1**

**For executing pipeline objects**

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.

**Description**

Pipeline objects

**Syntax**

```
pipelineobject.Start ( sourcetrans, destinationtrans, errorobject {, arg1, arg2,..., argn } )
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipelineobject</td>
<td>The name of a pipeline user object that contains the pipeline object to be executed</td>
</tr>
<tr>
<td>sourcetrans</td>
<td>The name of a transaction object with which to connect to the source database</td>
</tr>
<tr>
<td>destinationtrans</td>
<td>The name of a transaction object with which to connect to the target database</td>
</tr>
<tr>
<td>errorobject</td>
<td>The name of a DataWindow control or Data Store in which to store the pipeline error DataWindow</td>
</tr>
<tr>
<td>argn</td>
<td>One or more retrieval arguments as specified for the pipeline object in the Data Pipeline painter (optional)</td>
</tr>
</tbody>
</table>
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 or a Data Store for the pipeline-error DataWindow. Do not put a DataWindow object in the DataWindow control. The control displays 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 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 information about specifying the pipeline object in the Data Pipeline painter and how the settings affect committing, see the PowerBuilder Users 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.

Applies to

Timing objects
**Syntax**

```plaintext
timingobject.Start ( interval )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timingobject</td>
<td>The name of the timing object you want to activate.</td>
</tr>
<tr>
<td>interval</td>
<td>An expression of type double specifying the number of seconds that you want between timer events. The <em>interval</em> can be a whole number or fraction greater than 0 and less than or equal to 4,294,967 seconds. An interval of 0 is invalid.</td>
</tr>
</tbody>
</table>

**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. 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 is 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 occur before the Timer event.

The resolution of the interval depends on your operating system.

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 is not 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:

```powerscript
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:

```powerscript
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:

```powerscript
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:

```powerscript
long ll_hits

IF (isvalid(iuo_timing)) THEN
```
The Stop Timer button stops the timer, reenables the Start Timer button, and resets the hit counter:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
SharedObjectUnregister("Timing")
```

The `of_converterror` window function converts the ErrorReturn enumerated type to a string. It takes an argument of type ErrorReturn:

```powerbuilder
string ls_result
CHOOSE CASE a_error
CASE Success!
```

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:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
SharedObjectUnregister("Timing")
```

The `of_converterror` window function converts the ErrorReturn enumerated type to a string. It takes an argument of type ErrorReturn:

```powerbuilder
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 is 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.

Syntax
```
StartHotLink ( location, applname, topic {, bAnsi} )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>A string whose value is the location of the data in which a change of value triggers a HotLinkAlarm event. The format of the location depends on the application that contains the data.</td>
</tr>
<tr>
<td>applname</td>
<td>A string whose value is the DDE name of the server application.</td>
</tr>
</tbody>
</table>
### StartHotLink

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>topic</td>
<td>A string identifying the data or the instance of the application in which a change triggers a HotLinkAlarm event (for example, in Microsoft Excel, the topic name could be the name of an open spreadsheet).</td>
</tr>
<tr>
<td>bAnsi</td>
<td>(optional) A boolean identifying whether the string to get from the DDE server is in ANSI format. If bAnsi is NULL, false, or empty, PowerBuilder will first try to get the data in a UNICODE formatted string. If bAnsi is true, PowerBuilder will try to get the data in an ANSI formatted string.</td>
</tr>
</tbody>
</table>

**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`:

```plaintext
StartHotLink("Any", "MyPBApp", "Any")
```

This statement establishes a hot link with Microsoft Excel, which notifies the PowerBuilder window when the data at row 1 column 2 of `REGION.XLS` changes:

```plaintext
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 support.

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 references.
item (optional) | A comma-separated list of one or more strings (data within topic) that specify what your DDE server application supports (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 have 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 cannot 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.

Table 10-10: Events triggered by DDE requests and DDE functions available to each event

<table>
<thead>
<tr>
<th>Client action</th>
<th>Event triggered</th>
<th>Functions available</th>
<th>Purpose of function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sends a request for a hot link</td>
<td>RemoteHotLinkStart</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Sends a command to your application</td>
<td>RemoteExec</td>
<td>GetCommandDDE GetCommandDDEOrigin</td>
<td>Obtain the command Find out what client application sent the command</td>
</tr>
<tr>
<td>Sends data</td>
<td>RemoteSend</td>
<td>GetDataDDE GetDataDDEOrigin</td>
<td>Obtain the data Find out what client application sent the data</td>
</tr>
</tbody>
</table>
State

This statement causes your PowerBuilder application to begin acting as a server. It is known to other DDE applications as *MyPBApp*; its topic is *System*, and it supports items called *Table1* and *Table2*:

```
StartServerDDE(w_emp, "MyPBApp","System", &
"Table1", "Table2")
```

See also

StopServerDDE

State

Description

Determines whether an item in a ListBox control is highlighted.

Applies to

ListBox and PictureListBox controls

Syntax

```
listboxname.State( index )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>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</td>
</tr>
<tr>
<td>index</td>
<td>The number of the item for which you want to obtain the state</td>
</tr>
</tbody>
</table>

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 ListBox controls. SelectedItem reports the selection directly with no need for looping. In a multiple-selection ListBox control, 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:

```powerscript
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:

```powerscript
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:

```powerscript
IF lb_Contact.State(3) = 1 THEN ... 
```

See also

SelectedItem
SetState

---

**StepIt**

**Description**

Increments the current position in a progress bar control by the value specified in the SetStep property of the control.

**Applies to**

Progress bar controls

**Syntax**

```powerscript
control.StepIt() 
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>The name of the progress bar</td>
</tr>
</tbody>
</table>

**Return value**

Integer. Returns 1 if it succeeds and -1 if there is an error.
Usage
StepIt causes the position in a progress bar to wrap if the value of the SetStep takes the current position out of range. For example, if the SetStep value is 40, the current position 80, and the range is set from 0 to 100, the position on the redrawn progress bar after you call StepIt is 20.

The SetStep property can have a negative value. The default value for SetStep is 10.

Examples
This statement adds the SetStep increment to a progress bar control:

```
HProgressBar.StepIt();
```

See also
SetRange

Stop

Stop has two syntaxes.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deactivate a timing object</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Stop playing an animation</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

Syntax 1

For deactivating timing objects

Description
Deactivates a timing object.

Applies to
Timing objects

Syntax
```
timingobject.Stop();
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timingobject</td>
<td>The name of the timing object you want to deactivate</td>
</tr>
</tbody>
</table>

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
CHAPTER 10  PowerScript Functions

Syntax 2  For stopping an animation from playing
Description  Stops an animation (an AVI clip) from playing.
Applies to  Animation controls
Syntax  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animationname</td>
<td>The name of the animation control displaying the AVI clip</td>
</tr>
</tbody>
</table>

Return value  Integer. Returns 1 if it succeeds and -1 if the animation is not running or could not be stopped.
Usage  Use this function to stop an animation that is playing. A stopped animation can be restarted with the Play function.
Examples  This statement stops the AVI clip that is playing in the animation control MyAnimation:

```
MyAnimation.Stop()
```

See also  Play

StopHotLink

Description  Terminates a hot link with a DDE server application.

Caution  All arguments must match the arguments in an earlier StartHotLink call.

Syntax  

```
StopHotLink( location, applname, topic )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>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</td>
</tr>
<tr>
<td>applname</td>
<td>A string whose value is the DDE name of the server application, as specified in the StartHotLink function</td>
</tr>
<tr>
<td>topic</td>
<td>A string identifying the data or the instance of the application in which the hot link is stopped, as specified in the StartHotLink function</td>
</tr>
</tbody>
</table>
StopServerDDE

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

StopServerDDE

Description  Causes your application to stop acting as a DDE server application. Any subsequent requests from a DDE client application fail.

Syntax  `StopServerDDE ( { windowname, } applname, topic )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowname (optional)</td>
<td>The name of the server window. The default is the current window. If you have more than one server window, windowname is required.</td>
</tr>
<tr>
<td>applname</td>
<td>The DDE name for your PowerBuilder application.</td>
</tr>
<tr>
<td>topic</td>
<td>A string whose value is the topic you declared when you called StartServerDDE.</td>
</tr>
</tbody>
</table>

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:

```powerscript
StopServerDDE(w_emp, "MyPBApp", "System")
```

See also
StartServerDDE

String
String has two syntaxes.

<table>
<thead>
<tr>
<th>Syntax 1</th>
<th>For formatting data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td><code>String(data, {format})</code></td>
</tr>
<tr>
<td><strong>Argument</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><code>data</code></td>
<td>The data you want returned as a string with the specified formatting. Data can have a date, DateTime, numeric, time, or string datatype. Data can also be an Any variable containing one of these datatypes.</td>
</tr>
<tr>
<td><code>format</code> (optional)</td>
<td>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 datatype of data. If data is type string, format is required. The format can consist of more than one mask, depending on the datatype of data. Each mask is separated by a semicolon. (For details on each datatype, see Usage).</td>
</tr>
</tbody>
</table>
**String**

**Return value**

String. Returns `data` in the specified format if it succeeds and the empty string ("") if the datatype of `data` does not match the type of display mask specified, `format` is not a valid mask, or `data` is an incompatible datatype.

**Usage**

For date, DateTime, numeric, and time data, PowerBuilder uses the system’s default format for the returned string if you do not 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.

To display additional characters as part of the mask for a decimal value, you must precede each character with a backslash. For example, to display a decimal number with two digits of precision preceded by four asterisks, you must type a backslash before each asterisk:

```
dec{2} amount
string = ls_result
amount = 123456.32
ls_result = string(amount, "\*\*\*\*0.00")
```

The resulting string is ****123456.32.

For more information on specifying display formats, see the *PowerBuilder Users 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 does not match the datatype, PowerBuilder tries 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 do not specify a display format, the value is formatted as a DateTime value. This statement returns a string like "2/26/03 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 runtime DLLs available when the application is run. If you have installed localized runtime files in the development environment or on a user’s machine, then on that machine, the month in the resulting string is in the language of the localized files. For information about the localized runtime files, which are available in French, German, Italian, Spanish, Dutch, Danish, Norwegian, and Swedish, see the chapter on internationalization in Application Techniques.

Handling ANSI data  Since this function does not have an encoding argument to allow you to specify the encoding of the data, the string returned can contain garbage characters if the data has ANSI encoding. You can handle this by converting the ANSI string returned from the String function to a Unicode blob, and then converting the ANSI string in the blob to a Unicode string, using the encoding parameters provided in the Blob and String conversion functions:

```plaintext
ls_temp = String(long, "address")
lb_blob = blob(ls_temp)  //EncodingUTF16LE! is default
ls_result = string(lb_blob, EncodingANSI!)
```

Message object  You can also use String to extract a string from the Message object after calling TriggerEvent or PostEvent. For more information, see the TriggerEvent or PostEvent functions.

Examples  This statement applies a display format to a date value and returns Jan 31, 2002:

```plaintext
String(2002-01-31, "mmm dd, yyyy")
```

This example applies a format to the value in order_date and sets date1 to 6-11-02:

```plaintext
Date order_date = 2002-06-11
string datel
datel = 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:

```plaintext
Date order_date = 2002-06-11
string datel
SetNull(order_date)
datel = String(order_date, "m-d-yy;'none'")
```
This statement applies a format to a DateTime value and returns Jan 31, 2001 6 hrs and 8 min:

```powerbuilder
String(DateTime(2001-01-31, 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, 6-11-02 8:06 pm:

```powerbuilder
DateTime sys_datetime
string datetimel
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:

```powerbuilder
String(5, "$#,##0.00")
```

These statements set string1 to 0123:

```powerbuilder
integer nbr = 123
string string1
string1 = String(nbr, "0000;(000);****;empty")
```

These statements set string1 to (123):

```powerbuilder
integer nbr = -123
string string1
string1 = String(nbr, "000;(000);****;empty")
```

These statements set string1 to ****:

```powerbuilder
integer nbr = 0
string string1
string1 = String(nbr, "0000;(000);****;empty")
```

These statements set string1 to "empty":

```powerbuilder
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:

```powerbuilder
String("ABC", "@-@-@")
```
This statement returns A*B:

\[ \text{String} \left( \text{"ABC"}, \left[ \text{"@@@"} \right] \right) \]

This statement returns ABC:

\[ \text{String} \left( \text{"ABC"}, \left[ \text{"@@@"} \right] \right) \]

This statement returns a space:

\[ \text{String} \left( \text{"ABC"}, \left[ \text{" "} \right] \right) \]

This statement applies a display format to time data and returns 6 hrs and 8 min:

\[ \text{String} \left( 06:08:02, \left[ \text{"h "hrs and" m "min"} \right] \right) \]

This statement returns 08:06:04 pm:

\[ \text{String} \left( 20:06:04, \left[ \text{"hh:mm:ss am/pm"} \right] \right) \]

This statement returns 8:06:04 am:

\[ \text{String} \left( 08:06:04, \left[ \text{"h:mm:ss am/pm"} \right] \right) \]

See also

String method for DataWindows in the DataWindow Reference or online Help

**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**

\[ \text{String} \left( \text{blob} \left[ , \text{encoding} \right] \right) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blob</td>
<td>The blob whose value you want returned as a string. Blob can also be an Any variable containing a blob.</td>
</tr>
<tr>
<td>encoding</td>
<td>Character encoding of the blob you want converted. Values are:</td>
</tr>
<tr>
<td></td>
<td>• EncodingANSI!</td>
</tr>
<tr>
<td></td>
<td>• EncodingUTF8!</td>
</tr>
<tr>
<td></td>
<td>• EncodingUTF16LE! (default)</td>
</tr>
<tr>
<td></td>
<td>• EncodingUTF16BE!</td>
</tr>
</tbody>
</table>

**Return value**

String. Returns the value of blob as a string if it succeeds and the empty string (""") if it fails. If 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.
**String**

**Usage**

If the *encoding* argument is not provided, *String* converts a Unicode blob to a Unicode string. You must provide the *encoding* argument if the blob has a different encoding.

If the blob has a byte-order mark (BOM), *String* filters it out automatically. For example, suppose the blob’s hexadecimal display is: FF FE 54 00 68 00 69 00 73 00. The BOM is FF FE, which indicates that the blob has UTF-16LE encoding, and is filtered out. The string returned is “This”.

You can also use *String* to extract a string from the Message object after calling TriggerEvent or PostEvent. For more information, see the TriggerEvent or PostEvent functions.

**Examples**

This example converts the blob instance variable *ib_sblob*, which contains string data in ANSI format, to a string and stores the result in *sstr*:

```powerbuilder
string sstr
sstr = String(ib_sblob, EncodingANSI!)
```

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:

```powerbuilder
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**

Blob

*String* method for DataWindows in the *DataWindow Reference* or online Help
CHAPTER 10 PowerScript Functions

String_To_Object

Description

Gets an object reference based on a passed string.

This function is used by PowerBuilder clients connecting to EAServer.

Applies to

JaguarORB objects

Syntax

`jaguarorb.String_To_Object(objstring, object)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jaguarorb</code></td>
<td>An instance of JaguarORB.</td>
</tr>
<tr>
<td><code>objstring</code></td>
<td>A string that represents a CORBA object.</td>
</tr>
<tr>
<td><code>object</code></td>
<td>A variable of type CORBAObject that will contain the object reference.</td>
</tr>
</tbody>
</table>

Return value

Long. Returns 0 if it succeeds and a negative number if an error occurs.

Usage

The String_To_Object function allows you to instantiate a proxy instance without using the Jaguar naming service.

Connecting to EJB components

In PowerBuilder 7 and earlier releases, the JaguarORB String_To_Object function was used to access EJB components in EAServer. In PowerBuilder 8 and later, the Lookup function on the Connection object can be used to instantiate a proxy for the home interface of an EJB component in EAServer.

In PowerBuilder 9, the Lookup function on the EJBConnection PowerBuilder extension object can be used to instantiate proxies for EJB components running in any J2EE-compliant server.

When you use String_To_Object for proxy instantiation, you instantiate the object directly. The disadvantage of this approach is that you lose the benefits of server address abstraction that are provided by the naming service.

To use the naming service API explicitly, you can use the Resolve_Initial_References function to obtain an initial naming context. However, this technique is not recommended because it requires use of deprecated SessionManager::Factory methods. For more information about connecting to EAServer using the JaguarORB object, see Application Techniques.
The `String_To_Object` function can be used to obtain an EAServer authentication manager instance by using a URL format IOR. IOR strings in URL format must have the form:

```
protocol://host:iiop_port
```

where:

- `protocol` is `iiops` if connecting to a secure port and `iiop` otherwise
- `host` is the EAServer host address or machine name
- `iiop_port` is the port number for IIOP requests

An example of a URL-format IOR is:

```
iiop://hosta:2000
```

If the server is part of a cluster, the `objstring` argument can contain a list of IORs separated by semicolons.

After calling `String_To_Object`, you can use the Manager interface to obtain an instance of the Session interface, which allows you to create component instances. When you use the Manager and Session interfaces, you need to generate proxies for these interfaces and include these proxies in the library list for the client. For information about methods on these interfaces, see the interface repository documentation at the URL `http://yourhost:yourport/ir/`, where `yourhost` is the server's host name and `yourport` is the HTTP port number.

The `String_To_Object` function can also be used to deserialize a Proxy object reference. By serializing an object reference, you can save the state of the object so that it persists after the client terminates processing. Deserializing the object reference gets an object reference from a serialized string. `String_To_Object` is often used in conjunction with `Object_To_String`, which allows you to serialize an object reference.

**Examples**

The following example shows the use of the `String_To_Object` function to obtain an EAServer authentication manager instance. The function uses a URL format IOR:

```java
JaguarORB my_orb
CORBAObject my_corbaobj
Manager my_manager
Session my_session
Factory my_Factory
n_Bank_Account my_account

my_orb = CREATE JaguarORB
my_orb.init("ORBRetryCount=3, ORBRetryDelay=1000")
```
In this example, the component is an EJB component. When the _Narrow function is called to convert the object reference returned from the Lookup call on the Session object, the second argument includes the domain name as well as the package name. This is necessary if the Java package name uses the domainname.packagename format:

```
JaguarORB my_orb
CORBAObject my_corbaobj
Manager my_mgr
Session my_session
CartHome my_cartHome
Cart my_cart
long ll_return

my_orb = CREATE JaguarORB
my_orb.init("ORBLogFile='c:\temp\orblog'")
    my_corbaobj)
my_corbaobj._narrow(my_mgr, "SessionManager/Manager")
my_session = my_mgr.createSession("jagadmin", "")
my_corbaobj = my_session.lookup("Bank/n_Bank_Account")
my_corbaobj._narrow(my_Factory, "SessionManager/Factory")
my_corbaobj = my_Factory.create()
my_corbaobj._narrow(my_account, "Bank/n_Bank_Account")
my_account.withdraw(100.0)
```

See also
- Init
- Lookup
- _Narrow
- Object_To_String
- Resolve_Initial_References
SuspendTransaction

Description
SuspendTransaction suspends the EAServer transaction associated with the calling thread.

Applies to
CORBACurrent objects

Syntax
CORBACurrent.SuspendTransaction()

Argument | Description
---------|------------------
CORBACurrent | Reference to the CORBACurrent service instance

Return value
Unsigned long. Returns a handle that refers to the transaction associated with the thread or 0 if an error occurs.

Usage
The SuspendTransaction function returns a handle referring to the transaction associated with the calling thread. This handle can be passed to the ResumeTransaction function on the same or a different thread. When SuspendTransaction is called, the current thread is no longer associated with a transaction.

SuspendTransaction can be called by a client or a component that is marked as OTS style. It must be using the two-phase commit transaction coordinator (OTS/XA).

Examples
This example shows the use of the SuspendTransaction function to disassociate the calling thread from the current transaction:

```c++
// Instance variable:
// CORBACurrent corbcurr
integer li_rc
unsignedlong ll_handle

// Get and initialize an instance of CORBACurrent
... 
li_rc = corbcurr.BeginTransaction()
// do some transactional work
ll_handle = corbcurr.SuspendTransaction()
// do some nontransactional work
li_rc = corbcurr.ResumeTransaction(ll_handle)
// do some more transactional work
li_rc = corbcurr.CommitTransaction()
```

See also
BeginTransaction
CommitTransaction
GetTransactionName
ResumeTransaction
RollbackTransaction
SetTimeout
CHAPTER 10  PowerScript Functions

Synchronize

Starts synchronization between a remote and consolidated database. The syntax you use depends on whether you include command line parameters with the dbmlsync synchronization call.

<table>
<thead>
<tr>
<th>To start synchronization</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without including command line parameters</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>With command line parameters that you include in the synchronization call</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1  For synchronization without parameters

- **Description**: Starts synchronization between a remote and consolidated database.
- **Applies to**: MLSynchronization, MLSync controls
- **Syntax**: `SyncObject.Synchronize()`

#### Argument | Description
---|---
`SyncObject` | The name of the synchronization object.

- **Return value**: Integer. Returns 1 for success and -1 for failure. Any other return value is an error code from dbmlsync.
- **Examples**: If all the properties of a synchronization object are initialized, including userids and passwords, it is ready for immediate use. To launch a synchronization requires very little coding, as in the following example for an MLsync object named “nvo_my_mlsync”:

```power_script
nvu_my_mlsync mySync
Long rc
mySync = CREATE nvo_my_mlsync
mySync.Synchronize()
destroy mySync
```

You would typically add the above code to the Clicked event for a menu item or a command button on an application window.

### Syntax 2  For synchronization with parameters

- **Description**: Starts dbmlsync synchronization with command line parameters that are passed from the values of a syncparm structure.
- **Applies to**: MLSync controls

---

PowerScript Reference 1107
SyntaxFromSQL

### Syntax

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>syncObject</td>
<td>The name of the synchronization object.</td>
</tr>
<tr>
<td>cmdstring</td>
<td>A read-only string containing command line arguments for a synchronization call.</td>
</tr>
</tbody>
</table>

#### Return value

Integer. Returns 1 for success and any other value for failure.

#### Usage

The following is an example of a command string for a Synchronize call:

```plaintext
C:\Program Files\Sybase\SQL Anywhere 10\win32\dbmlsync.exe  
-c "DSN=salesdb_remote;UID=dba;PWD=sql" -n salesapi -u 50 -mp "pw10" -wc salesapi_50_sync -o "C:\temp\dbmlsync.log" -v+ -q -k
```

If the path to the dbmlsync executable (C:\Program Files\Sybase\SQL Anywhere 10\win32\dbmlsync.exe in the above example) is not part of the command string, the application searches the Windows registry to find it.

### SQL Anywhere 11

The path to dbmlsync.exe is changed for SQL Anywhere 11. It is in the Program Files\SQL Anywhere 11\bin32 or Program Files\SQL Anywhere 11\bin64 directory.

#### Examples

For MLSync objects, you can allow a user to edit the command line arguments for a synchronization call as follows:

```plaintext
long rc
string cmd
cmd = myMLSync.GetCommandString()
// Edit cmd however you wish
...
rc = myMLSync.Synchronize(cmd)
```

#### See also

CancelSync, GetCommandString, SetParm

---

**SyntaxFromSQL**

**Description**

Generates DataWindow source code based on a SQL SELECT statement.
CHAPTER 10  PowerScript Functions

Applies to  Transaction objects

Syntax  
\[ \text{transaction}.\text{SyntaxFromSQL} \left( \text{sqlselect}, \text{presentation}, \text{err} \right) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction</td>
<td>The name of a connected transaction object.</td>
</tr>
<tr>
<td>sqlselect</td>
<td>A string whose value is a valid SQL SELECT statement.</td>
</tr>
<tr>
<td>presentation</td>
<td>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 selected styles in the New DataWindow dialog box in the DataWindow painter. Keywords are: (Default) Tabular Grid Form (for freeform) Graph Group Label The Usage section lists the keywords you can use in presentation.</td>
</tr>
<tr>
<td>err</td>
<td>A string variable to which PowerBuilder will assign any error messages that occur.</td>
</tr>
</tbody>
</table>

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.

*Table owner in the SQL statement*  If the value of the LogID property of the Transaction object is not the owner of the table being accessed in the SQL statement for the SyntaxFromSQL function, then the table name in the SQL SELECT statement must be qualified with the owner name.
**Note for Adaptive Server Enterprise**
If your DBMS is Adaptive Server Enterprise and you call SyntaxFromSQL, PowerBuilder must determine whether the tables are updatable through a unique index. This is only possible if you set AutoCommit to true before calling SyntaxFromSQL, as shown here:

```sql
sqlca.autocommit=TRUE
ls_dws=sqlca-syntax-from-sql (sqlstmt, presentation, err)
sqlca.autocommit=FALSE
```

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:

```sql
"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:

```powerbuilder
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* contains 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:

```powerbuilder
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 method for DataWindows in the *DataWindow Reference* or online Help

Information on DataWindow object properties in the *DataWindow Reference*
SystemRoutine

Description
Provides the routine node representing the system root in a performance analysis model.

Applies to
Profiling object

Syntax

\[ \text{instancename}.\text{SystemRoutine} \left( \text{theroutine} \right) \]

Argument | Description
---|---
\text{instancename} | Instance name of the Profiling object.
\text{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:

```powerbuilder
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 } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tabcontrolname</code></td>
<td>The name of the Tab control for which you want to post events for its tab page user objects.</td>
</tr>
<tr>
<td><code>event</code></td>
<td>A value of the TrigEvent enumerated datatype 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 <code>tabcontrolname</code> and a script must exist for the event in <code>tabcontrolname</code>.</td>
</tr>
<tr>
<td><code>word</code> (optional)</td>
<td>A long value to be stored in the WordParm property of the system's Message object. If you want to specify a value for <code>long</code>, but not <code>word</code>, enter 0. (For cross-platform compatibility, WordParm and LongParm are both longs).</td>
</tr>
<tr>
<td><code>long</code> (optional)</td>
<td>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).</td>
</tr>
</tbody>
</table>

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

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 } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tabcontrolname</code></td>
<td>The name of the Tab control for which you want to trigger events for its tab page user objects.</td>
</tr>
<tr>
<td><code>event</code></td>
<td>A value of the TrigEvent enumerated datatype 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 <code>tabcontrolname</code> and a script must exist for the event in <code>tabcontrolname</code>.</td>
</tr>
<tr>
<td><code>word</code> (optional)</td>
<td>A long value to be stored in the WordParm property of the system’s Message object. If you want to specify a value for <code>long</code>, but not <code>word</code>, enter 0. (For cross-platform compatibility, WordParm and LongParm are both longs).</td>
</tr>
<tr>
<td><code>long</code> (optional)</td>
<td>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).</td>
</tr>
</tbody>
</table>

**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)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The angle (in radians) for which you want the tangent</td>
</tr>
</tbody>
</table>

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
ATan
Cos
Pi
Sin
Tan method for DataWindows in the DataWindow Reference or online Help

---

**Text**

Description
Obtains the text of an item in a ListBox control.

Applies to
ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

```
listboxname.Text(index)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>The name of the ListBox control in which you want the text of an item</td>
</tr>
<tr>
<td>index</td>
<td>The number of the item for which you want the text</td>
</tr>
</tbody>
</table>

Return value
String. Returns the text of the item in \( \text{listboxname} \) identified by \( \text{index} \). 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

\[
\text{current\_city}:
\]

```powershell
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**

\[
\text{editname}.\text{TextLine}()\]

**Argument** | **Description**  
--- | ---  
\text{editname} | The name of the DataWindow control, EditMask, MultiLineEdit, or RichTextEdit control 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:

```powershell
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
datatype of the source data.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract the time from DateTime data, or to extract a time stored in a blob</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Convert a string to a time</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Combine numbers for hours, minutes, and seconds into a time value</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

### Syntax 1: For DateTime and blob values

**Description**

Extracts a time value from a DateTime value or a blob.

**Syntax**

```
Time( datetime )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Return value**

Time. Returns the time in `datetime` as a time. If `datetime` does not contain a valid time or is an incompatible datatype, `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
```
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:

```powershell
time lt_time
lt_time = Time(BlobMid(ib_blob, 32))
```

See also Time method for DataWindows in the DataWindow Reference or online Help.

### Syntax 2

**For strings**

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 datatype, 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:

```powershell
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 \texttt{sle\textunderscore Time\textunderscore Received} to a time value:

\begin{verbatim}
\texttt{Time(sle\textunderscore Time\textunderscore Received\textunderscore Text)}
\end{verbatim}

\textbf{See also} Time method for DataWindows in the \textit{DataWindow Reference} or online Help

\section*{Syntax 3}

\textbf{For integers}

\textbf{Description} Combines integers representing hours, minutes, seconds, and microseconds into a time value.

\textbf{Syntax} \texttt{Time ( hour, minute, second \{, microsecond \} )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{hour}</td>
<td>The integer for the hour (00 to 23) of the time</td>
</tr>
<tr>
<td>\texttt{minute}</td>
<td>The integer for the minutes (00 to 59) of the time</td>
</tr>
<tr>
<td>\texttt{second}</td>
<td>The integer for the seconds (0 to 59) of the time</td>
</tr>
<tr>
<td>\texttt{microsecond}</td>
<td>(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)</td>
</tr>
</tbody>
</table>

\textbf{Return value} Time. Returns the time as a time datatype 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.

\textbf{Examples} These statements set \texttt{What\textunderscore Time} to a time value with microseconds, and display the resulting time as a string in \texttt{st\textunderscore 1}. The default display format does not include microseconds, so the \texttt{String} function specifies a display format with microseconds. Leading zeros are appended to the string value for microseconds:

\begin{verbatim}
Time What\textunderscore Time
What\textunderscore Time = Time(10, 15, 45, 234)
st\textunderscore 1\textunderscore Text = String(What\textunderscore Time, "hh:mm:ss:ffffff")
\end{verbatim}

The time in the string variable is set to 10:15:45:000234.

These statements set \texttt{What\textunderscore Time} to 10:15:45:

\begin{verbatim}
Time What\textunderscore Time
What\textunderscore Time = Time(10, 15, 45)
\end{verbatim}

\textbf{See also} Time method for DataWindows in the \textit{DataWindow Reference} or online Help
**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**
```powerbuilder
Timer( interval[, windowname] )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>interval</code></td>
<td>The number of seconds that you want between Timer events. interval can be a whole number or fraction greater than 0 and less than or equal to 4,294,967 seconds. If <code>interval</code> is 0, Timer turns off the timer so that it no longer triggers Timer events.</td>
</tr>
<tr>
<td><code>windowname</code></td>
<td>(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.</td>
</tr>
</tbody>
</table>

**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 retriggers 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.

**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:

```PowerScript
st_time.Text = String(Now(), "hh:mm")
```

See also

Idle

### ToAnsi

**Description**
Converts a character string to an ANSI blob.

**Syntax**

```
ToAnsi (string)
```

**Argument**  | **Description**  
--- | ---
string | A character string 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**
The `ToAnsi` function converts a Unicode character string to an ANSI blob. `ToAnsi` has the same result as `Blob(string, EncodingANSI)`, and it will be obsolete in a future version of PowerBuilder.

**Unicode file format**
Unicode files sometimes have two extra bytes at the start of the file to indicate that they are Unicode files. If you are opening a Unicode file in stream mode, skip the first two bytes if they are present.

See also

Blob
FromAnsi
FromUnicode
ToUnicode

### Today

**Description**
Obtains the system date and, in some cases, the system time.

**Syntax**

```
Today()
```

**Return value**
Date. Returns the current system date.
**Top**

**Usage**
Although the datatype 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 datatypes.

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 datatype 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, 2003:

```
IF Today() < 2003-04-15 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` method for DataWindows in the `DataWindow Reference` or online Help

---

**Top**

**Description**
Obtains the index number of the first visible item in a ListBox control. `Top` lets you to find out how the user has scrolled the list.

**Applies to**
ListBox and PictureListBox controls

**Syntax**

```
listboxname.Top ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listboxname</code></td>
<td>The name of the ListBox or PictureListBox in which you want the index of the first visible item in the list</td>
</tr>
</tbody>
</table>

**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()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listviewname</td>
<td>The name of the ListView control for which you want to find the number of columns</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns the number of columns if it succeeds and -1 if an error occurs.

Usage

Use when the ListView control is set to report view.

Examples

This example displays the number of columns in a ListView report view in a SingleLineEdit:

```
int li_cols
li_cols = lv_list.TotalColumns()
sl_info.text = "Total columns = " + string(li_cols)
```
TotalItems

See also  
TotalItems  
TotalSelected

TotalItems

Description  
Determines the total number of items in a ListBox control.

Applies to  
ListBox, DropDownListBox, PictureListBox, DropDownPictureListBox, and ListView controls

Syntax  
```
listcontrolname.TotalItems()
```

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 ListBox control that are selected.

Applies to  
ListBox, PictureListBox, and ListView controls

Syntax  
```
listcontrolname.TotalSelected()
```
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:

```powerbuilder
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:

```powerbuilder
IF lb_Actions.TotalSelected() > 3 THEN
    MessageBox("Warning", 
        "You can only select 3 items!")
ELSE
    ...
END IF
```

See also

TotalItems

---

**ToUnicode**

Description

Converts a character string to a Unicode blob.

Syntax

```
ToUnicode ( string )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A character string you want to convert to a Unicode blob</td>
</tr>
</tbody>
</table>

Return value

Blob. Returns a Unicode blob if it succeeds and an empty blob if it fails.

Usage

The ToUnicode function converts an ANSI character string to a Unicode blob. ToUnicode has the same result as Blob(string) and will be obsolete in a future version of PowerBuilder.
TraceBegin

**Unicode file format**
Unicode files sometimes have two extra bytes at the start of the file to indicate that they are Unicode files.

See also
FromAnsi
FromUnicode
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 drop-down list. It then begins logging the enabled activities for the first block of code to be traced:

```power
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**

```power
TraceClose ()
```

**Return value**

ErrorReturn. Returns one of the following values:

- **Success!** – The function succeeded
### TraceDisableActivity

**Description**
Disables logging of the specified trace activity.

**Syntax**

```
TraceDisableActivity ( activity )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| activity | A value of the enumerated datatype 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 |

**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
Return value

ErrorReturn. Returns one of the following values:

- **Success!** – The function succeeded
- **FileNotFoundException** – 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:

```powerscript
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activity</td>
<td>A value of the enumerated datatype TraceActivity that identifies the activity to be logged. Values are:</td>
</tr>
<tr>
<td></td>
<td>• ActError! – Occurrences of system errors and warnings</td>
</tr>
<tr>
<td></td>
<td>• ActESQL! – Embedded SQL statement entry and exit</td>
</tr>
<tr>
<td></td>
<td>• ActGarbageCollect! – Start and finish of garbage collection</td>
</tr>
<tr>
<td></td>
<td>• ActLine! – Routine line hits (if this value is enabled, ActRoutine! is automatically enabled)</td>
</tr>
<tr>
<td></td>
<td>• ActObjectCreate! – Object creation entry and exit</td>
</tr>
<tr>
<td></td>
<td>• ActObjectDestroy! – Object destruction entry and exit</td>
</tr>
<tr>
<td></td>
<td>• ActProfile! – Abbreviation for the ActRoutine!, ActESQL!, ActObjectCreate!, ActObjectDestroy, and ActGarbageCollect! values</td>
</tr>
<tr>
<td></td>
<td>• ActRoutine! – Routine entry and exit</td>
</tr>
<tr>
<td></td>
<td>• ActTrace! – Abbreviation for all activities except ActLine!</td>
</tr>
</tbody>
</table>

**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 drop-down list. Then it begins logging the enabled activities for the first block of code to be traced:

```plaintext
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 ()
```
**TraceError**

**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 calls 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:

```powerbuilder
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**

```powerbuilder
TraceError ( severity, message )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>severity</td>
<td>A long whose value is a number you want to indicate the severity of the error</td>
</tr>
<tr>
<td>message</td>
<td>A string whose value is the error message you want to add to the trace file</td>
</tr>
</tbody>
</table>
CHAPTER 10  PowerScript Functions

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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>A read-only string used to identify the trace file</td>
</tr>
<tr>
<td>timer</td>
<td>A value of the enumerated datatype TimerKind that identifies the timer. Values are:</td>
</tr>
<tr>
<td></td>
<td>- Clock! – Use the clock timer</td>
</tr>
<tr>
<td></td>
<td>- Process! – Use the process timer</td>
</tr>
<tr>
<td></td>
<td>- Thread! – Use the thread timer</td>
</tr>
<tr>
<td></td>
<td>- TimerNone! – Do not log timer values</td>
</tr>
</tbody>
</table>
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.

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.

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 drop-down list. Then it begins logging the enabled activities for the first block of code to be traced:

```power
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>A long whose value is a reference number you want to associate with the logged activity</td>
</tr>
<tr>
<td>message</td>
<td>A string whose value is the activity type value you want to add to the trace file</td>
</tr>
</tbody>
</table>

Return value ErrorReturn. This function always returns Success!.
**TriggerEvent**

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:

```powerbuilder
Try
  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
Catch
  "Handler code"
END Try
```

**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**

```powerbuilder
objectname.TriggerEvent({ event, word, long })
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectname</code></td>
<td>The name of any PowerBuilder object or control that has events associated with it.</td>
</tr>
<tr>
<td><code>event</code></td>
<td>A value of the TrigEvent enumerated datatype 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 <code>objectname</code> and a script must exist for the event in <code>objectname</code>.</td>
</tr>
</tbody>
</table>
**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 datatype, 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:

```plaintext
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 more information about events and when to use `PostEvent` and `TriggerEvent`, see `PostEvent`.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>word</code> (optional)</td>
<td>A long value to be stored in the WordParm property of the system’s Message object. If you want to specify a value for <code>long</code>, but not <code>word</code>, enter 0. (For cross-platform compatibility, WordParm and LongParm are both longs.)</td>
</tr>
<tr>
<td><code>long</code> (optional)</td>
<td>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).</td>
</tr>
</tbody>
</table>
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:

```plaintext
Send(Handle(Parent), 273, 0, Long(Handle(cb_OK), 0))
```

is equivalent to:

```plaintext
cb_OK.TriggerEvent(Clicked!)
```

**Examples**

This statement executes the script for the Clicked event in the CommandButton cb_OK immediately:

```plaintext
cb_OK.TriggerEvent(Clicked!)
```

This statement executes the script for the user-defined event cb_exit_request in the parent window:

```plaintext
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:

```plaintext
m_Appl.m_File.TriggerEvent(Clicked!)
```

**See also**

Post
PostEvent
Send
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:

```javascript
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.");
  }
}
```
This VBScript example calls the `TriggerPBEvent` function:

```vbscript
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, removeallspaces )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
<td>The string you want returned with leading and trailing spaces deleted</td>
</tr>
<tr>
<td><code>removeallspaces</code></td>
<td>A boolean indicating that all types of spaces should be deleted</td>
</tr>
</tbody>
</table>

---

1140
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.

If you do not include the optional removeallspaces argument or its value is false, only the space character (U+0020) is removed from the string.

If the removeallspaces argument is set to true, all types of space characters are removed. See LeftTrim for a list of space characters.

Examples
This statement returns BABE RUTH if all the leading and trailing spaces are space characters:

```
Trim(" BABE RUTH ")
```

This statement returns BABE RUTH if the leading and trailing spaces include other types of white space characters:

```
Trim(" BABE RUTH ", true)
```

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 method for DataWindows in the DataWindow Reference or online Help

**TrimW**

Description
Removes leading and trailing spaces from a string. This function is obsolete. It has the same behavior as Trim in all environments.

Syntax
```
TrimW ( string )
```

**Truncate**

Description
Truncates a number to the specified number of decimal places.
**Truncate**

**Syntax**

\[
\text{Truncate} \ (x, \ n)
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>The number you want to truncate.</td>
</tr>
<tr>
<td>(n)</td>
<td>The number of decimal places to which you want to truncate (x). Valid values are 0 through 28.</td>
</tr>
</tbody>
</table>

**Return value**

Decimal. Returns the result of the truncation if it succeeds and null if it fails or if any argument is null.

**Using Truncate on a computed field**

A real number loaded into a floating point register (used for calculation) is represented as precisely as the binary storage will permit. For example, the real number displayed as 2.07 is actually stored as 2.06999999999999999999999997.

Truncating such a number may not give the expected result. To avoid this problem, you can change the initial real datatype to long, integer, or decimal, or you can append a constant in the truncate argument:

\[
\text{Truncate} \ (x + 0.0000001, \ n)
\]

**Examples**

- This statement returns 9.2:
  \[
  \text{Truncate} \ (9.22, \ 1)
  \]
- This statement returns 9.2:
  \[
  \text{Truncate} \ (9.28, \ 1)
  \]
- This statement returns 9:
  \[
  \text{Truncate} \ (9.9, \ 0)
  \]
- This statement returns –9.2:
  \[
  \text{Truncate} \ (-9.29, \ 1)
  \]

**See also**

Ceiling
Int
Round
Truncate method for DataWindows in the *DataWindow Reference* or online Help
TrustVerify

Description
Called by EAServer when an SSL certificate chain needs to be approved for use by a client. This function is used by PowerBuilder clients connecting to EAServer.

Applies to
SSLCallBack objects

Syntax
sslcallback.TrustVerify (thesessioninfo, reason)

Return value
Long. Returns one of the following values:
1 TRUST_ONCE (accept the current connection)
2 TRUST_FAIL (reject the current connection)
3 TRUST_ALWAYS (accept and mark as trusted in the database)
4 TRUST_NEVER (reject and mark as untrusted in the database)
5 TRUST_SESSION (accept now and throughout the current session)
6 TRUST_FAIL_SESSION (reject throughout the current session)

Usage
A PowerBuilder application does not usually call the TrustVerify function directly. TrustVerify is called by EAServer when the internal SSL trust verification check fails to verify the server’s certificate chain or when the PIN to log in to the Sybase PKCS11 token was not supplied or incorrect. TrustVerify can be invoked when you are using any SSL protocol, because server authentication is a required step in the SSL handshake process.

To override the behavior of any of the functions of the SSLCallBack object, create a standard class user object that descends from SSLCallBack and customize this object as necessary. To let EAServer know which object to use when a callback is required, specify the name of the object in the callbackImpl SSL property. You can set this property value by calling the SetGlobalProperty function.

If you do not provide an implementation of TrustVerify, EAServer receives the CORBA::NO_IMPLEMENT exception and the connection is rejected.
To obtain a useful return value, provide the user with information about the reason for failure and ask the user to determine whether the server certificate chain can be trusted so that the session can continue. If the user specifies TRUST_FAIL or TRUST_ONCE, the function may be called again during the current session.

You can enable the user to cancel the attempt to connect by throwing an exception in this callback function. You need to catch the exception by wrapping the ConnectToServer function in a try-catch block.

This example checks whether the failure was called by a bad or missing PIN and returns TRUST_FAIL to call GetPin if it was. If not, it displays the reason why the server failed to verify the certificate chain and prompts the user to choose whether to continue with the session:

```powerbuilder
long       rc
string    stmp, stmp2
w_response w_ssl_response
string ls_rc

sslSessionInfo   mySessionInfo
rc = thesessioninfo._narrow(mySessionInfo, &"thesessioninfo")

is_tokenName = mySessionInfo.getProperty("tokenName")

CHOOSE CASE reason
CASE 4
  MessageBox("The SSL session requires a PIN", &
             "Please enter the PIN for access to the " + &
             is_tokenName + " certificate database.")
  return 2

CASE 5
  MessageBox("The PIN you entered is incorrect", &
             "Please reenter the PIN for access to the " + &
             is_tokenName + " certificate database.")
  return 2

CASE 1
  MessageBox("Certificate verification failed", &
             "Server's certificate chain is incomplete.ORB " &
             + "-nis unable to complete the chain using the " &
             + "CA certificates in the " &
             + "-nSybase PKCS11 Token.")
```
CASE 2
    MessageBox("Certificate verification failed", &
    "Server's certificate chain expired. One or " &
    + " more of the certificates in the " &
    + "chain is no longer valid.")
CASE 3
    MessageBox("Certificate verification failed", &
    "Server's certificate chain contains an " &
    + "unknown root certification authority. " &
    + "This CA is not found in the trust data in " &
    + "the Sybase PKCS11 Token.")
END CHOOSE

stmp += mySessionInfo.getProperty( "Version" )
stmp += "Host: "
stmp += mySessionInfo.getProperty( "host" )
stmp += "port: ":
  stmp += mySessionInfo.getProperty( "port" )
stmp += "ciphersuite: ":
  stmp += mySessionInfo.getProperty( "ciphersuite" )
stmp += "CertificateLabel: ":
  stmp += mySessionInfo.getProperty( "certificateLabel" )
stmp += "UserData: ":
  stmp += mySessionInfo.getProperty( "UserData" )
stmp += "tokenName: ":
  stmp += mySessionInfo.getProperty( "tokenName" )
stmp += "pkcs11Module: ":
  stmp += mySessionInfo.getProperty( "pkcs11Module" )

// Display information in a response window and return
// response with CloseWithReturn
openwithparm(w_response, stmp)
ls_rc = Message.StringParm
return long(ls_rc)
**TypeDef**

See also

- ConnectToServer
- GetCertificateLabel
- GetCredentialAttribute
- GetPin

**TypeDef**

**Description**

Determines the type of an object or control, reported as a value of the Object enumerated datatype.

**Applies to**

Any object

**Syntax**

\[\text{objectname}.\text{TypeOf()}\]

**Argument** | **Description**
---|---
\text{objectname} | The name of the object or control for which you want the type

**Return value**

Object enumerated datatype. Returns the type of \text{objectname}. If \text{objectname} is null, TypeOf returns null.

**Usage**

Use TypeOf to determine the type of a selected or dragged control.

**Examples**

If \text{dw_Customer} is a DataWindow control, this statement returns DataWindow!:

\[\text{dw_Customer}.\text{TypeOf()}\]

This example looks at the first five controls in the \text{w_dept} window’s Control array property. The loop executes some statements for each control that is a CheckBox:

```powerbuilder
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 \text{winobject} array the type of each object in the window’s Control array property:

```powerbuilder
object winobjecttype[]
long ll_count
```
FOR ll_count = 1 to UpperBound(Control[])
    winobjecttype[ll_count] = &
        TypeOf(Control[ll_count])
NEXT

If you do not know the type of a control passed via PowerObjectParm in the Message object, the following example assigns the passed object to a graphic object variable, the ancestor of all the control types, and assigns the type to a variable of type object, which is the enumerated datatype 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 check mark, if any, next to an item a drop-down or cascading menu and sets the item’s Checked property to false.

**Applies to**
Menu objects
Uncheck

Syntax

\textit{menuname.Uncheck()} \\
\textbf{Argument} \quad \textbf{Description} \\
\textit{menuname} \quad The fully qualified name of the menu selection from which you \hspace*{1em} want to remove the checkmark, if any. The menu must be on a drop- \hspace*{1em} down 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 \textit{menuname} is null, \hspace*{1em} \textit{Uncheck} returns null.

Usage

A checkmark next to a menu item indicates that the menu option is currently \hspace*{1em} on and that the user can turn the option on and off by choosing it. For example, \hspace*{1em} in the Window painter’s Design menu, a checkmark is displayed next to Grid \hspace*{1em} when the grid is on.

You can use \textit{Check} in an item’s Clicked script to mark a menu item when the \hspace*{1em} user turns the option on and \textit{Uncheck} to remove the check when the user turns \hspace*{1em} the option off.

\textbf{Equivalent syntax} \hspace{1em} You can set the object’s Checked property instead of \hspace*{1em} calling \textit{Uncheck}:

\textit{menuname.Checked = false}

This statement:

\begin{verbatim}
m_appl.m_view.m_grid.Checked = FALSE
\end{verbatim}

is equivalent to:

\begin{verbatim}
m_appl.m_view.m_grid.Uncheck()
\end{verbatim}

Examples

This statement removes the checkmark next to the \texttt{m_grid} menu selection in the \hspace*{1em} drop-down menu \texttt{m_view} on the menu bar \texttt{m_appl}:

\begin{verbatim}
m_appl.m_view.m_grid.Uncheck()
\end{verbatim}

This example checks whether the \texttt{m_grid} menu selection in the drop-down \hspace*{1em} menu \texttt{m_view} of the menu bar \texttt{m_appl} is currently checked. If so, the script \hspace*{1em} unchecks the item. If it is not checked, the script checks the item:

\begin{verbatim}
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
\end{verbatim}

See also

\textit{Check}
**Undo**

**Description**
Cancels the last edit in an edit control, restoring the text to the content before the last change.

**Applies to**
DataWindow, MultiLineEdit, RichTextEdit, and SingleLineEdit controls

**Syntax**
```
editname.Undo()```

**Argument** | **Description**
--- | ---
editname | The name of the DataWindow control, 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)```

**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()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectref</td>
<td>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: <code>dwcontrol.Object.dwobjectname</code></td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns 0 if it succeeds and -1 if an error occurs.

---

**UnitsToPixels**

**Argument**
- **units**: An integer whose value is the number of PowerBuilder units you want to convert to pixels
- **type**: A value of the ConvertType enumerated datatype 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:

```powerbuilder
integer Value
Value = UnitsToPixels(350, YUnitsToPixels!)
```

**See also**
PixelsToUnits
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 tries to find the file and if it fails it gives 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 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:

```powerscript
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:

```powerscript
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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string you want to convert to uppercase letters</td>
</tr>
</tbody>
</table>

**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` method for DataWindows in the *DataWindow Reference* or online Help

---

**UpperBound**

**Description**

Obtains the upper bound of a dimension of an array.

**Syntax**

```
UpperBound ( array [, n ] )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>The name of the array for which you want the upper bound of a dimension</td>
</tr>
<tr>
<td>n (optional)</td>
<td>The number of the dimension for which you want the upper bound. The default is 1</td>
</tr>
</tbody>
</table>

**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:

```
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[ ]  // Returns 0; no memory allocated
UpperBound(c)  // Returns 50
```

```
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 drop-down 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
END IF

See also

LowerBound

Which

Description

Allows a component to find out whether it is running on a transaction server.

Applies to

TransactionServer objects

Syntax

transactionserver.Which()

Argument | Description
--- | ---
transactionserver | Reference to the TransactionServer service instance

Return value

Integer. Returns 0 if the object is not running on a transaction server, 1 if it is running on EAServer, or 2 if it is running on COM+.

Usage

The Which function allows a custom class user object to perform different processing depending on its runtime context.
Examples

The code in the following example checks to see whether the runtime context is a transaction server (EAServer or COM+). If it is, it uses transaction semantics that are appropriate for a transaction server; otherwise, it uses COMMIT and ROLLBACK to communicate directly with the database:

```powerstation
// Instance variables:
// DataStore ids_datastore
// TransactionServer ts

Integer li_rc
long ll_rv

li_rc = this.GetContextService("TransactionServer", &
    ts)
IF li_rc <> 1 THEN
    // handle the error
END IF
...
...
ll_rv = ids_datastore.Update()

IF ts.WHICH() > 0 THEN
    IF ll_rv = 1 THEN
        ts.EnableCommit()
    ELSE
        ts.DisableCommit()
    END IF
ELSE
    IF ll_rv = 1 THEN
        COMMIT USING SQLCA;
    ELSE
        ROLLBACK USING SQLCA;
    END IF
END IF
```

See also

EnableCommit
IsInTransaction
IsTransactionAborted
Lookup
SetAbort
SetComplete
**WordCap**

**Description**
Capitalizes the first letter of each word in a passed script. It sets the remaining letters in each word to lowercase.

**Applies to**
All text objects

**Syntax**
```
WordCap ( text )
```

**Argument** | **Description**
--- | ---
`text` | String to be modified

**Return value**
String. If it succeeds, returns the text passed in the function argument with the first letter of each word in uppercase and the remaining letters in lowercase. Returns null if an error occurs.

**Examples**
This example takes user-entered text from a SingleLineEdit control, capitalizing the first letter in each word and setting the other letters to lowercase, before passing it in a string variable:

```powershell
string ls_fullname
ls_fullname = WordCap (sle_1.text)
```

The text `joe Macdonald` would be rendered as `Joe Macdonald` by the `WordCap` function.

---

**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 scroll bar, 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 scroll bar, 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 have placed controls). P_logo is the control that has 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 scroll bar, 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 scroll bar, or any toolbars on the left or right.

**Examples**
This example returns the width of the workspace area in the `w_employee` window:

```powerbuilder
ingenenture 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

\texttt{windowname.WorkSpaceX ()}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{windowname}</td>
<td>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</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns the distance that the left edge of the workspace area of \textit{windowname} is from the left edge of the screen (in PowerBuilder units). \texttt{WorkSpaceX} returns -1 if an error occurs. If \textit{windowname} is null, \texttt{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 scroll bar, if any) and the top and bottom of the window (not including the thickness of the frame or the title bar, menu bar, or horizontal scroll bar, if any).

Examples

This example returns the distance from the left edge of the screen to the left edge of the workspace area in the \texttt{w_employee} window:

\begin{verbatim}
integer workx
workx = w_employee.WorkSpaceX()
\end{verbatim}

See also

\texttt{PointerX}  
\texttt{PointerY}  
\texttt{WorkSpaceHeight}  
\texttt{WorkSpaceWidth}  
\texttt{WorkSpaceY}

\section*{WorkSpaceY}

\textbf{Description}

Obtains the distance between the top of a window’s workspace and the top of the screen.

For custom MDI frames, \texttt{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.

\textbf{Applies to}

Window objects

\textbf{Syntax}

\texttt{windowname.WorkSpaceY ()}
**Write**

**Description**
Writes data to an opened OLE stream object.

**Applies to**
OLEStream objects

**Syntax**

```powershell
olestream.Write ( dataforstream )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olestream</code></td>
<td>The name of an OLE stream variable that has been opened</td>
</tr>
<tr>
<td><code>dataforstream</code></td>
<td>A string, blob, or character array whose value you want to write to <code>olestream</code></td>
</tr>
</tbody>
</table>

**Return value**
Long. 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 oleststuff. Then it opens the stream called info in oleststuff and assigns it to the stream object olestr_info. It writes the contents of the blob variable lb_info to the stream olestr_info. Finally, it saves the storage oleststuff:

```powershell
boolean lb_memexists
OLEStorage olest_stuff
OLEStream olestr_info
integer li_result
long ll_result

olest_stuff = CREATE OLEStorage
li_result = olest_stuff.Open("c:\ole2\mystuff.ole")
IF li_result <> 0 THEN RETURN

li_result = olestr_info.Open(olest_stuff, "info", &
  stgReadWrite!, stgExclusive!)
IF li_result <> 0 THEN RETURN
ll_result = olestr_info.Write(lb_info)
IF ll_result = 0 THEN olest_stuff.Save()
```

See also

Length
Open
Read
Seek

**XMLParseFile**

**Description**

Parses an XML file and determines whether the file is well formed or complies with a specified grammar.

**Syntax**

```
XMLParseFile (xmlfilename {, validationscheme }{, parsingerrors }{,
  namespaceprocessing {, schemaprocessing {, schemafullchecking }}})
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmlstring</td>
<td>A string whose value is the name of the XML file to be parsed.</td>
</tr>
</tbody>
</table>
**XMLParseFile**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validationscheme</td>
<td>A value of the ValSchemeType enumerated datatype specifying the validation method used by the SAX parser. Values are:</td>
</tr>
<tr>
<td>(optional)</td>
<td>• ValNever! – Do not report validation errors.</td>
</tr>
<tr>
<td></td>
<td>• ValAlways! – Always report validation errors.</td>
</tr>
<tr>
<td></td>
<td>• ValAuto! – (default) Report validation errors only if a grammar is specified.</td>
</tr>
<tr>
<td>parsingerrors</td>
<td>A string buffer to which error messages can be saved. If not specified or set to null, errors display in a message box.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
</tr>
<tr>
<td>namespaceprocessing</td>
<td>A boolean specifying whether name space rules are enforced. When set to true, the parser enforces the constraints and rules defined by the W3C</td>
</tr>
<tr>
<td>(optional)</td>
<td>recommendation on namespaces in XML.</td>
</tr>
<tr>
<td></td>
<td>If validationscheme is set to ValAlways! or ValAuto!, the document must contain a grammar that supports the use of namespaces.</td>
</tr>
<tr>
<td></td>
<td>The default is false.</td>
</tr>
<tr>
<td>schemaprocessing</td>
<td>A boolean specifying whether schema support is enabled. When set to false, the parser does not process any schema found.</td>
</tr>
<tr>
<td>(optional)</td>
<td>If schemaprocessing is true, namespaceprocessing must also be set to true.</td>
</tr>
<tr>
<td></td>
<td>The default is false.</td>
</tr>
<tr>
<td>schemafullchecking</td>
<td>A boolean specifying whether schema constraints are checked. When set to true, the schema grammar is checked for errors.</td>
</tr>
<tr>
<td>(optional)</td>
<td>Setting schemafullchecking to true has no effect unless schemaprocessing is also set to true.</td>
</tr>
<tr>
<td></td>
<td>The default is false.</td>
</tr>
</tbody>
</table>

**Return value**

Long. Returns 0 for success and one of the following negative values if an error occurs:

-1 Parsing error

-2 Argument error

**Usage**

Use XMLParseFile to validate an XML file against a DTD or XML schema before proceeding with additional processing.

If no DTD or schema is included or referenced in the file, XMLParseFile checks whether the document contains well-formed XML. If the XML document fails validation or is not well-formed, XMLParseFile returns -1.
Because XSD You can also check the well-formedness of an XSD file because they are in XML format. The validation scheme must be ValAuto!, which is the default validation scheme.

To suppress the display of message boxes if errors occur, specify a string value for the parsingerrors argument.

The files pbxercesNN.dll and xerces-c_XX.dll, where NN represents the PowerBuilder version and XX represents the Xerces version, must be deployed with the other PowerBuilder runtime files in the search path of any application or component that uses this function.

Examples

These statements parse an XML document. If a DTD is included or referenced, the document is validated. Otherwise the parser checks for well-formedness. If the document passes validation, it is imported into a DataWindow control:

```powerbuilder
long ll_ret
ll_ret = XMLParseFile("c:\temp\mydoc.xml")
if ll_ret = 0 then dw_1.ImportFile("c:\temp\mydoc.xml")
```

These statements parse an XML document and save any errors in the string variable ls_err. If errors occur, no message boxes display. If a DTD is included or referenced, the document is validated. Otherwise the parser checks for well-formedness:

```powerbuilder
long ll_ret
string ls_err
ll_ret = XMLParseFile("c:\temp\mydoc.xml", ls_err)
```

These statements parse an XML document. If an XMLSchema is included or referenced, the document is validated, otherwise the parser checks for well-formedness:

```powerbuilder
long ll_ret
ll_ret = XMLParseFile("c:\temp\mydoc.xml", TRUE, TRUE)
```

These statements parse an XML document, validate against a given XML schema, and save any errors that occur in a string variable. If errors occur, no message boxes display. If no schema is included or referenced in the file, XMLParseFile returns -1:

```powerbuilder
long ll_ret
string ls_err
ll_ret = XMLParseFile("c:\temp\mydoc.xml", ValAlways!,
ls_err, TRUE, TRUE)
```
These statements parse an XML document, validate against a given XML schema, and parse the schema itself for additional errors. If no schema is included or referenced in the file, `XMLParseFile` returns -1:

```powerbuilder
long ll_ret
string ls_err
ll_ret = XMLParseFile("c:\temp\mydoc.xml", ValAlways!,
ls_err, TRUE, TRUE, TRUE)
```

These statements parse an XML document, validate against a given DTD, and save any errors that occur in a string variable. If errors occur, no message boxes display. If no DTD is included or referenced in the file, `XMLParseFile` returns -1:

```powerbuilder
long ll_ret
string ls_err
ll_ret = XMLParseFile("c:\temp\mydoc.xml", ValAlways!,
ls_err)
```

These statements parse an XSD file and test it for well-formedness. You must use `ValAuto!` when you parse an XSD file because there is no external schema associated with it. However, you do not need to specify the option when you call the function because it is the default validation method:

```powerbuilder
long ll_ret
ll_ret = XMLParseFile ("c:\mydoc.xsd")
```

See also

ImportFile
XMLParseString
ImportFile in the *DataWindow Reference* or online Help
**XMLParseString**  

Parses an XML string and determines whether the string is well formed or complies with a specified grammar.

**Syntax**

```plaintext
XMLParseString ( xmlstring {, validationscheme }{, parsingerrors } {, namespaceprocessing }{, schemaprocessing {}{, schemafullchecking }})
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xmlstring</code></td>
<td>A string that holds the XML document to be parsed.</td>
</tr>
</tbody>
</table>
| `validationscheme`     | A value of the ValSchemeType enumerated datatype specifying the validation method used by the SAX parser. Values are:  
  - `ValNever` – Do not report validation errors.  
  - `ValAlways` – Always report validation errors. Use `ValAlways` only when you know there is a DTD or schema against which the file can be validated.  
  - `ValAuto` – (default) Report validation errors only if a grammar is specified. |
| `parsingerrors`        | A string buffer to which error messages can be saved. If not specified or set to null, errors are shown to the user in a dialog box. |
| `namespaceprocessing`  | A boolean specifying whether name space rules are enforced. When set to `true`, the parser enforces the constraints and rules defined by the W3C recommendation on namespaces in XML.  
  If `validationscheme` is set to `ValAlways`! or `ValAuto`, the document must contain a grammar that supports the use of namespaces.  
  The default is `false`. |
| `schemaprocessing`     | A boolean specifying whether schema support is enabled. When set to `true`, the parser does not process any schema found.  
  If `schemaprocessing` is `true`, `namespaceprocessing` must also be set to `true`.  
  The default is `false`. |
| `schemafullchecking`   | A boolean specifying whether schema constraints are checked. When set to `true`, the schema grammar is checked for errors.  
  Setting `schemafullchecking` to `true` has no effect unless `schemaprocessing` is also set to `true`.  
  The default is `false`. |
XMLParseString

Return value  Long. Returns 0 for success and one of the following negative values if an error occurs:

-1  Parsing error
-2  Argument error

Usage  Use XMLParseString to validate an XML string against a DTD or XML schema before proceeding with additional processing.

If no DTD or schema is included or referenced in the string, XMLParseString checks whether the string contains well-formed XML. If the XML string fails validation or is not well-formed, XMLParseString returns -1.

XSD (schema) files are in XML format and you can check them for well-formedness. The validation scheme must be ValAuto!, which is the default validation scheme, because ValAlways! requires that there be a schema or DTD against which to validate the file.

For example, given the following schema file, the parser fails because there is no external XSD file that defines `xs:schema`, `xs:element`, and `xs:complexType`. The schema is defined by the namespace `http://www.w3.org/2001/XMLSchema`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="test3">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="test3_row" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Using ValAlways! also fails for an XML file if there is no schema defined or the reference does not point to a valid schema. If you use ValAuto!, validation is performed only if the schema or DTD file is present in the desired location. If it is not present, only well-formedness is checked.

To suppress the display of message boxes if errors occur, specify a string value for the `parsingerrors` argument.

The files `pbxercesNN.dll` and `xerces-c_XX.dll`, where `NN` represents the PowerBuilder version and `XX` represents the Xerces version, must be deployed with the other PowerBuilder runtime files in the search path of any application or component that uses this function.
Examples

These statements parse an XML string. If a DTD is included or referenced, the string is validated. Otherwise the parser checks for well-formedness:

```powerscript
// string argument as_xmlstring passed in
long ll_ret
ll_ret = XMLParseString(as_xmlstring)
```

These statements parse an XML string, validate against a given XML schema, and save any errors that occur in a string variable. If errors occur, no message boxes display. If no schema is included or referenced in the string, `XMLParseString` returns -1:

```powerscript
long ll_ret
string ls_xmlstr, ls_err
ll_ret = XMLParseString(ls_xmlstr, ValAlways!, ls_err, TRUE, TRUE)
```

These statements parse an XML string, validate against a given DTD, and save any errors that occur in a string variable. If errors occur, no message boxes display. If no DTD is included or referenced in the string, `XMLParseString` returns -1. If the string passes validation, it is imported into a DataWindow control:

```powerscript
long ll_ret
string ls_xmlstr, ls_err
ll_ret = XMLParseString(ls_xmlstr, ValAlways!, ls_err)
if ll_ret = 1 then dw_1.ImportString(ls_xmlstr)
```

See also

ImportString
XMLParseFile
ImportString in the DataWindow Reference or online Help
**Year**

**Description**
Determines the year of a date value.

**Syntax**

```powerbuilder
Year ( date )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>The date from which you want the year</td>
</tr>
</tbody>
</table>

**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 null.

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.

**Windows settings**

To make sure you get correct return values for the year, you must verify that `yyyy` is the Short Date Style for year in the Regional Settings of the user’s Control Panel. Your program can check this with the `RegistryGet` function.

If the setting is not correct, you can ask the user to change it manually or have the application change it (by calling the `RegistrySet` function). The user may need to reboot after the setting is changed.

**Examples**

This statement returns 2005:

```powerbuilder
Year (2005-01-31)
```

**See also**

Day
Month

`Year` method for DataWindows in the *DataWindow Reference* or online Help
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 cannot 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 is not 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
Yield
Index

Symbols
! (enumerated value) 30
& see ampersand
* (multiplication) 70
+ (addition) 70
++, += (assignment shortcuts) 122
/ (division) 70
// (comments) 4
/= (assignment shortcut) 122
; (SQL) 15
< (less than) 72
<= (less than or equal) 72
<> (not equal) 72
= (assignment) 40
= (relational) 72
> (greater than) 72
>= (greater than or equal) 72
? (dynamic SQL) 172, 173, 176
^ (exponentiation) 70
_Is_A function 670
_Narrow function 762
~ (assignment shortcut) 122
- see dashes
~ see tilde
' see quotes

A
Abs function 326
absolute value 326
access levels
functions 60
group label 46
variables 43
ACos function 326
Activate event 186
Activate function 327
active sheet 787
active window 828
Adaptive Server Enterprise 1110
AddCategory function 329
AddColumn function 330
AddData function 331, 351
AddItem function 333
addition operator 70
AddLargePicture function 338
AddPicture function 339
address keyword 1137
address, mail 727, 736, 737
AddSeries function 340
AddSmallPicture function 341
AddStatePicture function 342
AddToLibraryList function 343
AllowEdit property 963
ampersand (&) 15
ancestor
calling function or event 114
hierarchy 370
objects 84
return values from events 115
script, calling 123
AncestorReturnValue variable 115
AND operator 72
angle
calculating arccosine 326
calculating arcsine 348
calculating arctangent 349
calculating cosine 404
calculating sine 1076
calculating tangent 1115
converting to/from radians 823, 824
animation
starting 825
stopping 1095
ANSI, string conversion 500, 501, 1121, 1125
Any datatype 26
API and database handles 424
application
  closing DDE channel 381
  connecting to 392, 393, 396
  elapsed time 405
  exporting object as syntax 705
  handle 505, 614
  listing objects 701
  posting messages 838
  recreating objects from syntax 706
  restarting 912
  retrieving arguments 387
  running 925
  server 1091, 1096
  terminating 137
  yielding to 1169
application name 1090, 1091, 1096
Application objects, SetTransPool function 1063
Arabic functions
  IsAllArabic 672
  IsAnyArabic 673
  IsArabic 674
  IsArabicAndNumbers 675
arccosine 326
arcsine 348
arctangent 349
arguments
  command line 387
  for events 184
  functions and events 106
  hot link 1089, 1096
  server application 1091, 1097
arithmetic operators 70
Arrange function 344
ArrangeOpen enumerated datatype 787
ArrangeSheets function 345
ArrangeTypes enumerated datatype 345
array functions
  LowerBound 725
  UpperBound 1152
arraylists 56
arrays
  about 48
  assigning values 54, 56, 120
  chars and strings 78
  copying 120
  default values 51
errors 57
example 347
initializing 56
input parameter for dynamic SQL 1006
mailRecipient 726
message ID 728
passing as arguments 108
stream 884, 1160
variable-size 52
arrow pointer 1024
Asc function 346
AscA function 346
ASCII values
  converting characters to 346
  of nonprinting characters 868
ASin function 348
assignment
  arrays 51, 54, 56
  overflow 77
  shortcut operators 122
  statements 120
asterisk in text patterns 745
ATan function 349
AttachmentFile property 735
audio (beep) 349
AutoCommit 1110
Autoinstantiate setting 85
automation 981, 982, 983
axis, graphs
  categories 329, 361, 426, 640
  inserting data 644

B
back quote 123
background color, graphs
  data points 533, 997
  series 592, 1039
background layer of DataWindow 1027
backslash in text patterns 744
backspace, specifying 7
bands, DataWindow, moving objects to 1027
BAT file 925
batch applications 839
beam pointer 1024
Index

Beep function 349
BeginDownload event 186
BeginDrag event 187
BeginLabelEdit event 190
BeginLogScan event 192
BeginRightDrag event 192
BeginSync event 195
BeginTransaction function 350
BeginUpload event 195
birth dates 1168
bitmaps
  assigning to picture control 1023
  in rich text 662
  printing 851
  retrieving from clipboard 374
blob datatype 21
Blob function 351, 352
blob functions
  Blob 351, 352
  BlobEdit 353
  BlobMid 354
  Len 697, 699
BlobEdit function 353
BlobMid function 354
blobs
  assigning to picture control 1023
  converting 509
  converting to byte arrays 509
  converting to bytes 358, 508
  converting to string 351, 1098
  declaring 37
  extracting values from 416, 420, 425, 664, 717, 719, 886, 1117
  inserting data into 353
  reading streams into 884
  selecting from database 162
  updating 165
  writing to stream 1160
boolean datatype 21
border
  determining distance from 826, 827
  printing 862, 865, 866
bottom layer of DataWindow 1027
bound 725, 1152
brackets in text patterns 744

BuildModel function 355
byte array
  converting to blob 352
  obtaining from blob 509
byte datatype 22
Byte function 358

C
C functions
  decoding returned values 667
  passing values to 717
CALL statement
  about 123
  not using 184
Cancel button 751
Cancel function 359
cancellation
  allowing 1169
  of edits 1149
  of pipeline object 359
  of printing 852
CancelSync function 360
CanUndo function 360
capitalization
  in category names 329, 640
  in series names 340
  lowercase 725
  uppercase 1152
caret in text patterns 744
carriage return
  in INI files 880
  specifying 7
cascaded windows, arranging sheets 345
cascading opened windows 787
case sensitivity, comparisons 72
categories, graphs
  adding data values to series 329, 331
  adding to a series 329
  clicked 765
  counting 361
  deleting 426, 903
  identifying 361, 362
  importing data 620, 623, 627
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertCategory function</td>
<td>329</td>
</tr>
<tr>
<td>CategoryCount function</td>
<td>361</td>
</tr>
<tr>
<td>CategoryName function</td>
<td>362</td>
</tr>
<tr>
<td>Ceiling function</td>
<td>363</td>
</tr>
<tr>
<td>ChangeDirectory function</td>
<td>363</td>
</tr>
<tr>
<td>ChangeMenu function</td>
<td>364</td>
</tr>
<tr>
<td>Channel, DDE</td>
<td>380, 785</td>
</tr>
<tr>
<td>CharA function</td>
<td>366</td>
</tr>
<tr>
<td>CharW function</td>
<td>365</td>
</tr>
<tr>
<td>Check function</td>
<td>367</td>
</tr>
<tr>
<td>Check function</td>
<td>367</td>
</tr>
<tr>
<td>Checked property</td>
<td>1147</td>
</tr>
<tr>
<td>CollapseItem function</td>
<td>386</td>
</tr>
<tr>
<td>COLUMNClick event</td>
<td>206</td>
</tr>
<tr>
<td>ClassDefinition objects, FindMatchingFunction</td>
<td>494</td>
</tr>
<tr>
<td>ClassName function</td>
<td>370</td>
</tr>
<tr>
<td>Clear function</td>
<td>372</td>
</tr>
<tr>
<td>ClearBoldDates function</td>
<td>373</td>
</tr>
<tr>
<td>Clearing text</td>
<td>372</td>
</tr>
<tr>
<td>Clicked event</td>
<td>196, 767</td>
</tr>
<tr>
<td>Clipboard contents as replacement text</td>
<td>901</td>
</tr>
<tr>
<td>copying</td>
<td>401</td>
</tr>
<tr>
<td>cutting</td>
<td>411</td>
</tr>
<tr>
<td>importing data from</td>
<td>620</td>
</tr>
<tr>
<td>pasting and linking</td>
<td>820</td>
</tr>
<tr>
<td>pasting from</td>
<td>818</td>
</tr>
<tr>
<td>retrieving and replacing contents</td>
<td>374</td>
</tr>
<tr>
<td>Clipboard function</td>
<td>374</td>
</tr>
<tr>
<td>CLOSE Cursor statement</td>
<td>149</td>
</tr>
<tr>
<td>Close event</td>
<td>202, 376, 912</td>
</tr>
<tr>
<td>Close function</td>
<td>376</td>
</tr>
<tr>
<td>CLOSE Procedure statement</td>
<td>150</td>
</tr>
<tr>
<td>CloseChannel function</td>
<td>380</td>
</tr>
<tr>
<td>CloseQuery event</td>
<td>204, 377</td>
</tr>
<tr>
<td>CloseTab function</td>
<td>381</td>
</tr>
<tr>
<td>CloseWithReturn function</td>
<td>383</td>
</tr>
<tr>
<td>Closing</td>
<td></td>
</tr>
<tr>
<td>DDE channel</td>
<td>380</td>
</tr>
<tr>
<td>print job</td>
<td>854</td>
</tr>
<tr>
<td>windows</td>
<td>376</td>
</tr>
<tr>
<td>Code generating DataWindow</td>
<td>1108</td>
</tr>
<tr>
<td>object</td>
<td>705</td>
</tr>
<tr>
<td>reusing</td>
<td>840</td>
</tr>
<tr>
<td>Cold link</td>
<td>455, 584, 785, 1033</td>
</tr>
<tr>
<td>CollapseItem function</td>
<td>386</td>
</tr>
<tr>
<td>Colors</td>
<td></td>
</tr>
<tr>
<td>and edit masks</td>
<td>1015</td>
</tr>
<tr>
<td>data point</td>
<td>533, 906, 996</td>
</tr>
<tr>
<td>red, green, and blue components of</td>
<td>916</td>
</tr>
<tr>
<td>series</td>
<td>591, 1039</td>
</tr>
<tr>
<td>supported</td>
<td>552</td>
</tr>
<tr>
<td>Table of standard colors</td>
<td>916</td>
</tr>
<tr>
<td>ColumnClick event</td>
<td>206</td>
</tr>
<tr>
<td>Columns</td>
<td></td>
</tr>
<tr>
<td>determining insertion point position</td>
<td>832</td>
</tr>
<tr>
<td>in list</td>
<td>642</td>
</tr>
<tr>
<td>pasting text into</td>
<td>819</td>
</tr>
<tr>
<td>COM file</td>
<td>925</td>
</tr>
<tr>
<td>Command line, retrieving arguments</td>
<td>387</td>
</tr>
<tr>
<td>CommandParm function</td>
<td>387</td>
</tr>
</tbody>
</table>
commands
  getting from DDE client 514
  receiving from DDE application 911
comments
  in library 700
  using 4
COMMIT statement 151
CommitTransaction function 388
comparing
  numbers 664, 747, 756
comparing strings 72
computer
  beeping 349
  reporting CPU time 405
concatenation operator 73
condensed mode 869
conditional compilation 17
configuration settings
  reading 878, 880
  saving 1029
CONNECT statement 152
Connection objects
  ConnectToServer function 398
  CreateInstance function 408
  DisconnectServer function 444
connections, to OLE object 392
ConnectMobiLink event 207
ConnectToNewRemoteObject function 392
ConnectToServer function 398
constants
  assigning values 40
  declaring 47
  where to declare 34
Constructor event 208
ContextInformation objects
  GetCompanyName function 517
  GetFixesVersion function 561
  GetHostObject function 565
  GetMajorVersion function 571
  GetMinorVersion function 573
  GetName function 574
  GetShortName function 598
  GetVersionName function 613
ContextKeyword objects, GetContextKeywords function 518
continuation character 15
CONTINUE statement 126
continuous line style
  setting for data points 999
  setting for series 1041
Control array 801, 803
count, of data points in a series 413
CPU
  getting information about 552
  time 405
Cpu function 405
CREATE statement 127, 828
CreateDirectory function 405
CreateInstance function 408
CreatePage function 410
cross mouse pointer 1024
crosstabs, creating from source code 1108
Index

current row and scrolling 944, 946
sheet 787
cursor
custom 1025
displaying pop-up menus 828
print 847
cursors, database
closing 149
declaring 148, 153
opening 160
custom class user objects 85
Cut function 411
cutting, to clipboard 411

dash line style
about 999, 1042
setting for series 1042
dashes, prohibiting in variable names 5
DashesInIdentifiers option 5
data
adding to a graph series 331, 333
clearing 902
converting to type long 716, 718
correcting pipeline 897
finding in DataWindow 483
from OLE server 526
getting DDE 528
importing 620
inserting into a blob 353
obtaining from control 523
receiving from DDE application 911
sending to DDE client 992
sharing 414
to OLE server 991
transferring 1082
writing to file 478, 480
writing to stream 1160
data expressions, Any datatype 28
Data Pipeline painter 359, 1083
data points
adding to a scatter graph 333
clicked 765
deleting 426
inserting 643
reporting appearance of 532
reporting explosion percent 531
resetting colors 906
setting style 996
value of 523, 539
data types
literals 23
database stored procedures 145
databases
canceling changes 160
commiting changes 151
connecting to 152
cursor, opening 160
deleting rows 155, 156
disconnecting from 156
fetching rows 158
handle 424
inserting rows 159
on restart 912
repairing 897
selecting rows 161
transactions 1063
transferring data between 1082
updating 164
updating cursored row 166
DataChange event 209
DataSource function 414
datatype checking and conversion functions
Asc 347
Char 365, 366
Date 415
DateTime 419
Dec 425
Double 445
Integer 664
IsDate 678
IsNull 682
IsNumber 683
IsTime 686
Long 716, 718
Real 886
String 1097
Time 1117
datatype mappings, EAServer 31
Index

datatypes
about 21
assignment 77
blob 351, 352
byte 358, 509
date 418
determining 370
effect of operators 75
enumerated 30
external functions 62
literals 22, 23, 24, 26, 76
mismatch when pasting 819
numeric 75
promotion 75
promotion for function arguments 104
real 886
setting to NULL 1019
standard 21
string 1097
system object 29
time 1117
unknown 26
windows 769
DataWindow control
data expressions and Any datatype 28
for pipeline errors 1083
DataWindow functions
CanUndo 360
CategoryCount 361
CategoryName 362
Clear 372
Clipboard 374
Copy 401
Cut 411
DataCount 413
FindCategory 485
FindNext 496
FindSeries 497
GetData 523
GetDataPieExplode 531
GetDataStyle 532
GetSeriesStyle 590
LineCount 708
ObjectAtPointer 765
Paste 818
PasteRTF 821
Position 833
ReplaceText 901
ResetDataColors 906
Scroll 942
SelectedLength 954
SelectedLine 956
SelectedStart 958
SelectedText 959
SelectText 966
SeriesCount 975
SeriesName 976
SetDataPieExplode 995
SetDataStyle 996
SetPosition 1027
SetSeriesStyle 1039
TextLine 1116
Undo 1149
DataWindow object
creating from SELECT statement 1108
deleting from libraries 701
exporting as syntax 705
listing 701
recreating from syntax 706
date datatype 22
Date function 415
date, day, and time functions
Day 420
DayName 421
DayNumber 422
DaysAfter 423
Hour 617
Minute 756
Month 759
Now 765
RelativeDate 893
RelativeTime 894
Second 948
SecondsAfter 949
Today 1121
Year 1168
DateChanged event 209
dates
checking string 678
converting to 417
DateTime datatype 416, 419
day of week 421, 422
determining interval 423
getting dynamic 544, 546
in blobs 416
obtaining current 1121
obtaining day of month 420
DateSelected event 210
DateTime datatype 22
DateTime function 419
Day function 420
DayName function 421
DayNumber function 422
DaysAfter function 423
dBase file, importing data from 623, 627
DBError event 210
DBHandle function 424
DBNotification event 212
dbsign 148
DDE channel
closing 380
requesting data 585
DDE client functions
CloseChannel 380
ExecRemote 455
GetDataDDE 528
GetDataDDEOrigin 528
GetRemote 584
OpenChannel 785
RespondRemote 911
SetRemote 1032
StartHotLink 1089
StopHotLink 1095
DDE server functions
GetCommandDDE 514
GetCommandDDEOrigin 515
GetDataDDE 528
GetDataDDEOrigin 528
RespondRemote 911
SetDataDDE 992
StartServerDDE 1091
StopServerDDE 1096
DDL, executing through dynamic SQL 171, 172
Deactivate event 215
DEBUG symbol 17
DebugBreak function 425
Dec function 425
decimal datatype
about 23
converting to 425
declaring 37
declarations
access levels 43
arrays 48
constants 47
expressions as initial values 41
external functions 57
syntax 37
variables 33
where to declare 34
DECLARE Cursor statement 153
DECLARE Procedure statement 153
definition, font for printing 856
DELETE statement 155
DELETE Where Current of Cursor statement 156
DeleteAllItems event 215
DeleteCategory function 426
DeleteColumn function 427
DeleteColumns function 427
DeleteData function 428
DeleteItem event 216
DeleteItem function 429
DeleteLargePicture function 432
DeleteLargePictures function 432
DeletePicture function 433
DeletePictures function 433
DeleteSeries function 434
DeleteSmallPicture function 435
DeleteSmallPictures function 435
DeleteStatePicture function 436
DeleteStatePictures function 436
descendant
determining class of 370
opening user object 794, 795, 802, 804
opening window 773
return values from events 115
DESTROY statement
about 130
ending a mail session 732
DestroyModel function 437
Destructor event 217, 381, 382
detail bands, moving objects to 1027
diagonal fill pattern 1001, 1043
dialog
Insert Object 661
Open File 553
PasteSpecial 822
Save File 558
diamond fill pattern 1001, 1043
dimension 725
dimension of array 1152
directory, of library 701, 703
DirectoryExists function 437
DirList function 438
DirSelect function 440
Disable function 441
DisableCommit function 442
DISCONNECT statement 156
DisconnectMobiLink event 217
DisconnectObject function 443
DisconnectServer function 444
display format, applying to string 1097
DisplayMessage event 218
distributed applications
ConnectToServer function 398
DisconnectServer function 444
SharedObjectDirectory function 1065
SharedObjectGet function 1065
SharedObjectRegister function 1069, 1070
division 757
division operator 70, 71
DLL files, executing functions from 64
DLLs for external functions 60
document windows 787
dollar sign in text patterns 744
dot notation
about 36
instance variables 35
structures 80
dotted line style
setting for data points 999
setting for series 1042
double colon 123
double datatype 23
Double function 445
DoubleClicked event 219
DoubleParm property 790, 797, 799, 806, 808
DoVerb function 446
Drag function 447
DragDrop event 222
DragEnter event 227
DraggedObject function 448
dragging, TreeView items 1004
DragLeave event 228
DragObject functions
ClassName 370
Drag 447
Hide 616
Move 760
PointerX 826
PointerY 827
PostEvent 839
Print 845
Resize 908
SetFocus 1007
SetPosition 1026
SetRedraw 1032
Show 1070
TriggerEvent 1136
TypeOf 1146
DragWithin event 229
Draw function 450
drawing objects
and SetFocus function 1007
posting events 839
setting color of 916
DrawObject functions
ClassName 370
Hide 616
Move 760
Print 845
Resize 908
Show 1070
TypeOf 1146
DropDown event 232
DropDownListBox control, deleting text 372
DropDownListBox functions
AddItem 334
Clear 372
Copy 401
Cut 411
DeleteItem 429
DirList 438
DirSelect 440
DraggedObject 448
Index

FindItem 488
InsertItem 649
Paste 818
Position 833
Post 838
ReplaceText 901
Reset 902
SelectedLength 954
SelectedStart 958
SelectedText 959
SelectItem 961
SelectText 966
Text 1115
TotalItems 1124

DropDownPictureListBox functions
AddItem 335
AddPicture 339
Clear 372
Copy 401
Cut 411
DeletePicture 433
DeletePictures 433
FindItem 488
InsertItem 650
Paste 818
Position 833
ReplaceText 901
SelectedLength 954
SelectedStart 958
SelectedText 959
SelectItem 961
SelectText 966
Text 1115
TotalItems 1124

DWOObjects, OLE functions 327, 401, 446, 1150
dynamic calls
about 99
ersors 101
dynamic libraries 343, 1013
dynamic library (DLL) 1089
dynamic SQL
about 167
considerations 169
DynamicDescriptionArea 168
DynamicStagingArea 168
Format 1 171
Format 2 172
Format 3 173
Format 4 176
formats listed 167
NULL values 172, 174
ordering statements 169
preparing DynamicStagingArea 169
statements 168
dynamic SQL functions
GetDynamicDate 544
GetDynamicDateTime 546
GetDynamicNumber 548
GetDynamicString 550
GetDynamicTime 551
SetDynamicParm 1005
DynamicDescriptionArea
about 168
properties 177
DynamicStagingArea
about 168
preparing 169

E
EAServer datatype mappings 31
edit control
counting lines in 708
deleting text from 372
determining insertion point position 832
inserting clipboard contents 374
replacing text 901
selected text 954, 958
EditLabel function 406, 451
EditMask functions
CanUndo 360
Clear 372
Copy 401
Cut 411
GetData 525
LineCount 708
LineLength 709
Paste 818
Position 833
ReplaceText 901
Index

Scroll 942
SelectedLength 954
SelectedLine 956
SelectedStart 958
SelectedText 959
SelectText 966
SetMask 1015
TextLine 1116
Undo 1149
embedded SQL 145
Enable function 452
EnableCommit function 453
Enabled property 616, 1032
EndDownload event 232
EndLabelEdit event 233
EndLogScan event 234
EndSync event 235
EndUpload event 236
EntryList function 454
enumerated datatypes 30
envelope, mail message header 734
environment
  getting information about 552
  TEMP variable 735
error checking
  cascaded calls 110
  compiling scripts 100
Error DataWindow 897
Error event 236
error handling
  after SQL statements 147
  calling functions or events 101, 103
error objects, creating 127
ErrorMessage event 239
errors
  displaying pipeline 1083
  runtime 71
escape sequences 868
events
  about 91, 183
  adding to queue 839
  ancestor 114
  and hidden objects 616
  and print jobs 854
  arguments 106, 184
  cascaded calls 110, 112
defined 92
errors when calling 101
extending 105
finding 95
overriding 105
posting 96, 110, 1113
return codes 184
return values 109, 184
similarities to functions 93
static and dynamic 98
system 92, 183
triggering 96, 184, 1114, 1136
user-defined 183, 185
exclamation point icon 751
exclusive share mode 777, 780
ExecRemote function 455
executable
  returning application handle 614
  running 925
EXECUTE statement 157, 1005
execution errors 100
EXIT statement 133
Exp function 458
ExpandAll function 459
ExpandItem function 459
exponent 458
exponentiation operator 70
expressions
  Any datatype 27
  checking for NULL 682
datatype promotion 75
datatypes 75
DataWindows and Any datatype 28
in declaration 41
literals 76
operators and datatypes 75
external functions 57
ExternalException event 240

F
Fact function 460
FETCH statement 158
file functions
  FileClose 460
Index

FileDelete 462
FileExists 464
FileLength 465, 466
FileOpen 468
FileRead 471
FileReadEx 473
FileSeek 475, 477
FileWrite 478
FileWriteEx 480
GetFileOpenName 553
GetFileSaveName 558
FileClose function 460
FileCopy function 461
FileDelete function 462
FileEncoding function 463
FileExists event 243
FileExists function 464
FileLength function 465
FileLength64 function 466
FileMessage event 244
FileMove function 467
FileOpen function 468
FileRead function 471
FileReadEx function 473
files
  importing data from 623
  linking 711
  security and sharing violation 465, 466
FileSeek function 475, 477
FileWrite function 478
FileWriteEx function 480
Fill function
  about 482
  and printing 482
FillA function 483
FillPattern 535, 1000, 1043
FillW function 482, 483
filtering filenames 553, 558
Find function 483
FindCategory function 485
FindClassDefinition function 486
FindFunctionDefinition function 487
FindItem function 488
FindMatchingFunction function 494
FindNext function 496
FindSeries function 497
FindTypeDefinition function 498
flicker 1032
focus
  and line length 709
  finding control with 562
  selected text 955, 958, 959, 967
  setting 1007
folder 701
fonts
  and string length when printing 876
  defining for printing 856
  FontFamily enumerated datatype 856
  FontPitch enumerated datatype 856
  names and sizes 857
  setting 870
  when printing 847
  when printing DataWindow controls 855
footer, moving objects to 1027
foreground color
  data points 533, 997
  series 592, 1039
foreground layer of DataWindow 1027
Form presentation style 1108
formats, applying to strings 1097
formfeed, specifying 7
frame window 828, 1158, 1159
FromAnsi function 500
FromUnicode function 501
function object
  exporting as syntax 705
  listing 701
  re-creating from syntax 706
functions
  about 91
  access level for external 60
  ancestor 114
  arguments 106
  calling global and system 112
  cascaded calls 110, 112
  case sensitivity 112
  chars as arguments 78
  DLLs 60
  errors when calling 101
  external 57
  external datatypes 62
  external, defined 92
Index

external, mail 731
external, reporting database handle 424
finding 94
overloading 104
overriding 104
posting 96, 110
return values 109
similarities to events 93
static and dynamic 98
system, defined 92
triggering 96
type promotion 104
user-defined 92

G

garbage collection 84, 128, 130
GarbageCollect function 502
GarbageCollectGetTimeLimit function 502
GarbageCollectSetTimeLimit function 503
Gesture event 244, 293
GetActiveSheet function 504
GetAlignment function 505
GetApplication function 505
GetArgElement function 506
GetAutomationNativePointer function 507
GetByte function 508
GetByteArray function 509
GetCertificateLabel function 509
GetChildrenList function 511
GetColumn function 513
GetCommandDDE function 514
GetCommandDDEOrigin function 515
GetCommandString function 516
GetCompanyName function 517
GetContextKeywords function 518
GetContextService function 519
GetCredentialAttribute function 520
GetCurrentDirectory function 523
GetData function 523
GetDataDDE function 528
GetDataDDEOrigin function 528
GetDataLabelling function 530
GetDataPieExplose function 531
GetDataStyle function 532
GetDataTransparency function 538
GetDataValue function 539
GetDateLimits function 541
GetDbmlsyncPath function 542
GetDisplayRange function 543
GetDynamicDate 177
GetDynamicDate function 544
GetDynamicDateTime function 177
GetDynamicDateTime function 546
GetDynamicDecimal 177
GetDynamicDecimal function 548
GetDynamicNumber 177
GetDynamicNumber function 549
GetDynamicString 177
GetDynamicString function 550
GetDynamicTime 177
GetDynamicTime function 551
GetEnvironment function 552
GetFileOpenName function 553
GetFileSaveName function 558
GetFirstSheet function 560
GetFixesVersion function 561
GetFocus event 246
GetFocus function 562
GetFolder function 563
GetGlobalProperty function 564
GetHostObject function 565
GetItem function 566
GetItemAtPointer function 569
GetLastReturn function 569
GetLibraryList function 570
GetMajorVersion function 571
GetMinorVersion function 573
GetName function 574
GetNativePointer function 575
GetNextSheet function 576
GetObjectRevisionFromRegistry function 577
GetOrigin function 578
GetParagraphSetting function 579
GetParent function 579
GetPin function 581
GetRecordSet function 583
GetRemote function 584
GetSelectedDate function 587
GetSelectedRange function 588
GetSeriesLabelling function 589
Index

<table>
<thead>
<tr>
<th>Function/Operator</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetSeriesStyle function</td>
<td>590</td>
</tr>
<tr>
<td>GetSeriesTransparency function</td>
<td>597</td>
</tr>
<tr>
<td>GetShortName function</td>
<td>598</td>
</tr>
<tr>
<td>GetStatus function</td>
<td>599</td>
</tr>
<tr>
<td>GetSyncRegistryProperties function</td>
<td>601</td>
</tr>
<tr>
<td>GetText function</td>
<td>602</td>
</tr>
<tr>
<td>GetToday function</td>
<td>605</td>
</tr>
<tr>
<td>GetToolbar function</td>
<td>605</td>
</tr>
<tr>
<td>GetToolbarPos function</td>
<td>607, 1057</td>
</tr>
<tr>
<td>GetTransactionName function</td>
<td>610</td>
</tr>
<tr>
<td>GetURL function</td>
<td>611</td>
</tr>
<tr>
<td>GetValue function</td>
<td>612</td>
</tr>
<tr>
<td>GetVersionName function</td>
<td>613</td>
</tr>
<tr>
<td>global functions</td>
<td></td>
</tr>
<tr>
<td>calling</td>
<td>112</td>
</tr>
<tr>
<td>defined</td>
<td>92</td>
</tr>
<tr>
<td>global scope operator</td>
<td>35</td>
</tr>
<tr>
<td>global variables</td>
<td></td>
</tr>
<tr>
<td>about</td>
<td>34</td>
</tr>
<tr>
<td>scope operator</td>
<td>35</td>
</tr>
<tr>
<td>GOTO statement</td>
<td>136</td>
</tr>
<tr>
<td>Graph functions</td>
<td></td>
</tr>
<tr>
<td>AddCategory</td>
<td>329</td>
</tr>
<tr>
<td>AddData</td>
<td>331, 351</td>
</tr>
<tr>
<td>AddSeries</td>
<td>340</td>
</tr>
<tr>
<td>CategoryCount</td>
<td>361</td>
</tr>
<tr>
<td>CategoryName</td>
<td>362</td>
</tr>
<tr>
<td>Clipboard</td>
<td>375</td>
</tr>
<tr>
<td>DataCount</td>
<td>413</td>
</tr>
<tr>
<td>DeleteCategory</td>
<td>426</td>
</tr>
<tr>
<td>DeleteData</td>
<td>428</td>
</tr>
<tr>
<td>DeleteSeries</td>
<td>434</td>
</tr>
<tr>
<td>FindCategory</td>
<td>485</td>
</tr>
<tr>
<td>FindSeries</td>
<td>497</td>
</tr>
<tr>
<td>GetData</td>
<td>523</td>
</tr>
<tr>
<td>GetDataPieExplode</td>
<td>531</td>
</tr>
<tr>
<td>GetDataStyle</td>
<td>532</td>
</tr>
<tr>
<td>GetSeriesStyle</td>
<td>590</td>
</tr>
<tr>
<td>GetImportClipboard</td>
<td>620</td>
</tr>
<tr>
<td>GetImportFile</td>
<td>623</td>
</tr>
<tr>
<td>GetImportString</td>
<td>627</td>
</tr>
<tr>
<td>InsertCategory</td>
<td>640</td>
</tr>
<tr>
<td>InsertData</td>
<td>643</td>
</tr>
<tr>
<td>InsertSeries</td>
<td>663</td>
</tr>
<tr>
<td>ModifyData</td>
<td>757</td>
</tr>
<tr>
<td>Reset</td>
<td>903</td>
</tr>
<tr>
<td>SaveAs</td>
<td>931</td>
</tr>
<tr>
<td>SeriesCount</td>
<td>975</td>
</tr>
<tr>
<td>SeriesName</td>
<td>976</td>
</tr>
<tr>
<td>SetDataPieExplode</td>
<td>995</td>
</tr>
<tr>
<td>SetDataStyle</td>
<td>996</td>
</tr>
<tr>
<td>SetSeriesStyle</td>
<td>1039</td>
</tr>
<tr>
<td>graphics, printing</td>
<td>851</td>
</tr>
<tr>
<td>graphs</td>
<td></td>
</tr>
<tr>
<td>categories</td>
<td>331</td>
</tr>
<tr>
<td>overlay</td>
<td>596</td>
</tr>
<tr>
<td>series</td>
<td>340</td>
</tr>
<tr>
<td>grColorType enumerated datatype</td>
<td>533</td>
</tr>
<tr>
<td>grDataType enumerated datatype</td>
<td>524, 539, 597, 1003, 1047</td>
</tr>
<tr>
<td>Grid presentation style</td>
<td>1108</td>
</tr>
<tr>
<td>gObjectType enumerated datatype</td>
<td>766</td>
</tr>
<tr>
<td>Group presentation style</td>
<td>1108</td>
</tr>
<tr>
<td>grResetType enumerated datatype</td>
<td>903</td>
</tr>
<tr>
<td>grSymbolType enumerated datatype</td>
<td>1043</td>
</tr>
</tbody>
</table>

**H**

HALT statement 137

handle

database 424
DDE 380, 785, 1091
mailSession object 731, 973
validating 688

Handle function 614

header band, moving objects to 1027

Hebrew functions

IsAllHebrew 672
IsAnyHebrew 674
IsHebrew 678
IsHebrewAndNumbers 679

height

object 908
workspace 1156

Help

calling Winhelp 1072
displaying MicroHelp 1017

Help event 247

Help Search window 1072

hidden objects 1070

Hide event 248
Index

Hide function  616
hierarchies
  child items in a list  654, 656, 659
  items in TreeView  386, 459
  sorting  1079
  sorting children  1078
  system  29, 370
high word of long  667
highlighting
  items in lists  961, 1092
  scrolling  946
  setting  1049
horizontal fill pattern  1001, 1043
horizontal scrollbar for lists  334
horizontal scrolling, when adding items to lists  334
host variables in SQL  146
hot link
  about  992
  determining origin of  528
  determining source of data  529
  establishing  1089
  terminating  1095
HotLinkAlarm event  249
Hour function  617
hourglass pointer  1024
HyperlinkToURL function  617
hyphens, prohibiting in variable names  5

I
icons
  arranging in ListView  344
  arranging windows  345
  in message box  751
identifier names, rules for  5
Idle event  249
IDs for events  183
IF...THEN statement
  about  138
  multiline  139
  single-line  138
image
  assigning to picture control  1023
  retrieving from clipboard  374
ImpersonateClient function  619
ImportClipboard function  620
ImportFile function  623
importing, data  623, 627
ImportString function  627
inbox
  deleting messages from  728
  downloading messages to  733
  reading mail messages  734
  retrieving message IDs from  728, 729
  saving messages in  739
IncomingCallList function  629
index
  highlight state of  1049, 1092
  obtaining top  1122
  of listbox item  953, 962
indicator variables in SQL  146
Inet objects
  GetURL function  611
  HyperlinkToURL function  617
  PostURL function  841
Information icon  751
inheritance  84
  back quote  123
  double colon  123
  PowerBuilder objects  29
INI file
  reading  878, 880
  writing values to  1029
Init function  631, 632
InkEdit functions
  RecognizeText  887
InkPicture functions
  LoadInk  712
  LoadPicture  713
  ResetInk  907
  ResetPicture  907
  Save  927
  SaveInk  940
input fields in rich text  634, 635, 636, 637, 638
InputFieldChangeData function  634
InputFieldCurrentName function  635
InputFieldDeleteCurrent function  636
InputFieldGetData function  637
InputFieldInsert function  638
InputFieldLocate function  638
InputFieldSelected event  250
Index

Insert Object dialog 661
INSERT statement 159
InsertCategory function 640
InsertClass function 642
InsertColumn function 642
InsertData function 643
InsertFile function 647
inserting strings 898, 902
insertion point
  character position 952
  in editable controls 709
  in text line 956, 1116
  when pasting from clipboard 818
InsertItem event 250
InsertItem function 648
InsertItemFirst function 654
InsertItemLast function 656
InsertItemSort function 659
InsertObject function 661
InsertPicture function 662
InsertSeries function 663
instance variables
  about 34
  class of 370
  dot notation 35
  initialized 42
instances
  checking if valid 688
  defined 80
  of user object 793, 796, 801, 805
Int function 664
integer
  combining into long value 716, 718
  converting to 664
  converting to char 365, 366
  obtaining from blob 664
integer datatype 23
Integer function 664
Intel 552
InternetData function 666
InternetRequest objects, InternetData function 666
interpersonal messages 729
interprocess messages 729
interval 1120
IntHigh function 667
IntLow function 667
InvokePBFunction function 668
Is_A ( _Is_A ) function 670
IsAlive function 671
IsAllArabic function 672
IsAllHebrew function 672
IsAnyArabic function 673
IsAnyHebrew function 674
IsArabic function 674
IsArabicAndNumbers function 675
IsCallerInRole function 676
IsDate function 678
IsHebrew function 678
IsHebrewAndNumbers function 679
IsImpersonating function 680
IsInTransaction function 681
IsNull function 682
IsNumber function 664, 683
IsPreview function 684
IsSecurityEnabled function 685
IsTime function 686
IsTransactionAborted function 687
IsValid function
  about 688
  and Handle function 614
  description 688
  getting active sheet 504
  getting open sheets 560, 576
ItemActivate event 251
ItemChanging event 253
ItemCollapsed event 254
ItemCollapsing event 255
ItemExpanded event 256
ItemExpanding event 257
ItemPopulate event 258
items
  adding to lists 333, 648
  deleting from list 429, 902
  determining number of selected 1124
  determining total number of 1124
  highlight state of 1049, 1092
  index number of 953
  linking 711
  selecting 961
  text of 953, 1115
  top 1061, 1122
J
JaguarORB, initializing 631
JavaScript keywords, reserved 10

K
Key event 258
keyboard
determining key pressed 688
selecting text 402
KeyCode enumerated datatype
about 688
values 689
KeyDown function 689
keywords 10

L
Label presentation style 1108
labels for GOTO 6
language for OLE automation 981, 983
LastPos function 692
Layer enumerated datatype 345
Layered window 790
layering opened windows 787
layout 855
Left function 694, 695
LeftA function 694
LeftTrim function 695, 696
LeftW function 694, 695
Len function 697, 699
LenA function 698
length
text 709
OLE stream 699
selected text 954
string or blob 697, 699
Length function 699
LibDirType enumerated datatype 701
LibDirType enumerated datatype 703
LibExportType enumerated datatype 705
libraries
deleting objects from 701, 703
pasting and linking object from 820
search path 343, 570, 1013
Library functions
LibraryCreate 700
LibraryDelete 701
LibraryDirectory 701
LibraryDirectoryEx 703
LibraryExport 705
LibraryImport 706
LibraryCreate function 700
LibraryDelete function 701
LibraryDirectory function 701
LibraryDirectoryEx function 703
LibraryExport function 705
LibraryImport function 706
limit, numeric 363
line
setting 872
when printing text 847
LineCount function 708
LineDown event 260
LineLeft event 261
LineLength function 709
LineList function 710
LineRight event 262
lines
and SetFocus function 1007
color for data points 533
counting number of 708
determining length 709
graphs, color for data points 997
graphs, color for series 592, 1039
graphs, style for data points 535, 998
graphs, style for series 593, 594, 1041
printing 860, 874
scrolling 942
selected text 956
spacing in rich text 599
text 1116
width 535
LineUp event 263
linking
clipboard contents 820, 822
establishing 711
LinkTo function 711
ListBox functions
AddItem 334
Index

DeleteItem 429
DirList 438
DirSelect 440
FindItem 488
InsertItem 649
Reset 902
SelectedIndex 953
SelectedItem 953
SelectItem 961
SetState 1049
SetTop 1061
State 1092
Text 1115
Top 1122
TotalItems 1124
TotalSelected 1124

lists
adding items 648
adding new item 333
deleting items from 902
horizontal scrollbar 334
of files in listbox 438
of objects in libraries 701, 703
sorted 334
ListView control, columns 1010
ListView functions
AddColumn 330
AddItem 336, 337
AddLargePicture 338
AddSmallPicture 341
AddStatePicture 342
Arrange 344
DeleteColumn 427
DeleteColumns 427
DeleteItem 430
DeleteLargePicture 432
DeleteLargePictures 432
DeleteSmallPicture 435
DeleteSmallPictures 435
DeleteStatePicture 436
DeleteStatePictures 436
EditLabel 451
FindItem 489, 491
GetColumn 513
GetItem 567
GetOrigin 578
InsertColumn 642
InsertItem 651
ListView 1123
SelectedItem 953
SetItem 1009
SetOverlayPicture 1020
Sort 1078
TotalItems 1124
TotalSelected 1124

literals
datatypes of 76
specifying 22, 23, 24, 26
LoadInk function 712
LoadPicture function 713
local variables 34
Log function
about 715
inverse 714
natural logarithm 714
logarithms 714, 715
logical operators 72
LogTen function
about 715
inverse 715
long datatype
about 23
converting to 716, 718
returning high word 667
returning low word 667
Long function 716, 718
longlong datatype 23
LongParm
posting events 839
specifying values for 717
triggering events 1137
Lookup function 720
LOOP 131
LOOP, in DO...LOOP statement 131
loops
about 131
iterative 134
leaving 133
skipping current iteration 126
yielding within 1169
LoseFocus event 264, 752
low word of long 667
Lower function 725
LowerBound function 725
lowercase 725

M

mail functions
  mailAddress 727
  mailDeleteMessage 728
  mailGetMessages 729
  mailHandle 731
  mailLogoff 731
  mailLogon 732
  mailReadMessage 734
  mailRecipientDetails 736
  mailResolveRecipient 737
  mailReturnCode 733
  mailSaveMessage 739
  mailSend 742
mailAddress function 726
mailDeleteMessage function 728
mailHandle function 731
mailLogoff function 731
mailLogon function 732
mailLogonOption enumerated datatype 733
mailReadMessage function 734
mailReadOption enumerated datatype 735
mailRecipient structure 737
mailRecipientDetails function 736
mailResolveRecipient function 737
mailReturnCode function 733
mailSaveMessage function 739
mailSend function 742
main window 760
MAPI 731
margins 847, 869, 1021
masks
  applying to strings 1097
  matching 743
  reporting length of 709
  setting 1015
Match function 743
Max function 747
maximum value below a limit 664
maximum value of two numbers 747

MDI Client (MDI_1) functions
  ClassName 370
  Hide 616
  Print 845
  Resize 908
  SetRedraw 1032
  Show 1071
  TypeOf 1146

MDI frame
  arranging windows 345
  changing menus 364
  displaying pop-up menus 828
  getting active 504
  opening sheets 771, 787, 789
  specifying MicroHelp text 1017

MDI frame functions
  ArrangeSheets 345
  GetActiveSheet 504
  GetFirstSheet 560
  GetNextSheet 576
  GetToolbar 605
  GetToolbarPos 607, 1057
  OpenSheet 787
  OpenSheetWithParm 789
  Print 845
  SetMicroHelp 1017
  SetToolbar 1055
measurement 1149
member, OLE 748, 749, 750
MemberDelete function 748
MemberExists function 749
MemberRename function 750
memory
  allocation for arrays 52
  and variable-sized arrays 1152
  releasing after mail session 732

Menu functions
  Check 367
  ClassName 370
  Disable 441
  Enable 452
  PopMenu 828
  Show 1070
  TriggerEvent 1136
  TypeOf 1146
  Uncheck 1147
Menu objects
- exporting as syntax 705
- listing 701
- recreating from syntax 706

menus
- changing 364
- Checked property 367
- creating object 127
- displaying 828
- for sheet 787

message ID array 729

Message object
- accessing parameters 809
- and TriggerEvent function 1137
- close return value 383
- creating 127
- determining type 1147
- extracting strings from 1099, 1102
- open sheet parameters 789
- PowerObjectParm property 384
- properties 797, 799, 806, 808
- specifying values for 717

MessageBox function 751, 861

messages
- deleting 728
- posting 838
- saving 739, 742
- sending to a window 973

metacharacters 743, 744

MicroHelp 1017

Microsoft Windows
- and DDE 584
- and timers 1120
- calling Winhelp 1072
- defining fonts for printing 857
- displaying Save File response window 558
- events and messages in 840
- getting filenames 553
- getting information about 552
- message numbers 973
- obtaining handle 614
- returned messages 667
- RightToLeft version 672, 673, 674, 675, 678, 679, 914

Mid function 753

MidA function 755

MidW function 753, 755

Min function 756

minimum value
- above a limit 363
- of two numbers 756

Minute function 756

miscellaneous functions
- IsValid 688
- KeyDown 689
- MessageBox 824
- PixelsToUnits 824
- RGB 916
- SetNull 1019
- SetPointer 1024
- TypeOf 1146
- UnitsToPixels 1149

Mod function 757

Modified event 265

ModifyData function 757

modulus 757

monitor 552

Month function 759

month, obtaining the day of 420

More Windows menu item 788

mouse
- selecting text 402
- setting shape of pointer 1024

MouseDown event 267

MouseMove event 269

MouseUp event 272

Move function 760

Moved event 275

multidimensional arrays 50, 54

MultiLineEdit functions
- CanUndo 360
- Clear 372
- Copy 401
- Cut 411
- LineCount 708
- LineLength 709
- Paste 818
- Position 833
- ReplaceText 901
- Scroll 942
- SelectedLength 954
- SelectedLine 956
Index

SelectedStart 958
SelectedText 959
SelectText 966
TextLine 1116
Undo 1149
multiplication operator 70, 71
MultiSelect property
  highlighted state 1049, 1095
  selecting items 953, 954, 963

N
  names, rules for 5
  naming conventions 39
  Narrow ( _Narrow ) function 762
  negative numbers 1075
  nested OLE objects 778, 781
  newline, specifying 7
  NEXT, in FOR...NEXT statement 134
  NextActivity function 763
  NOT operator 72
  Notify event 276
  Now function 765
  null object references 791, 797, 799, 806, 808, 811, 813
  NULL values
    about 8
    checking 682
    dynamic SQL 174
    in boolean expressions 72
    setting variables to 1019
    testing for 8
  numbers
    category 362
    checking string 683
    comparing 747, 756
    converting char 365, 366, 417, 425
    determining maximum 363
    determining sign of 1075
    getting dynamic 548, 549
    logarithm of 714, 715
    multiplying by pi 824
    of day of week 422
    of lines, counting 708
    random 881, 882
    returning remainder 757
    rounding 923
    truncating 1141
  numeric functions
    Abs 326
    ACos 326
    ASin 348
    ATan 349
    Ceiling 363
    Cos 404
    Exp 458
    Fact 460
    Int 664
    Log 714
    Max 747
    Min 756
    Mod 757
    Pi 824
    Rand 881
    Randomize 882
    Round 923
    Sign 1075
    Sin 1076
    Sqrt 1081
    Tan 1115
    Truncate 1141
  N-Up presentation style 1108

O
  ObjectAtPointer function 765
  objects
    about 80
    ancestor 84
    assignment 86
    changing position 1027
    creating instance 127
    deleting from libraries 701
    destroying instance 130
    determining class of 370
    determining type 1146
    garbage collection 84, 130
    general references 11
    hiding 616, 760
    inserting 642, 647, 661
Index

instantiating 83
linking 711
loading 343, 1013
moving 760
obtaining handle 614
parent object 579
passing as arguments 106
posting events 839
recreating 706
redrawing 1032
reference handle 80
saving OLE 928
selecting 964
setting focus 1007
triggering events 1136
under pointer 765

objects, Connection
ConnectToServer function 398
CreateInstance function 406, 408
DisconnectServer function 444

objects, shared
SharedObjectDirectory function 1065
SharedObjectGet function 1065
SharedObjectRegister function 1069
SharedObjectUnregister function 1070

ObjectToString function 768
OffsetPos function 769
Offsite enumerated datatype 328
OK button 751

OLE DWObject functions
Activate 327
Copy 401
DoVerb 446
UpdateLinksDialog 1150

OLE expressions and Any datatype 28

OLEControl functions
Activate 327
Clear 372
Copy 401
Cut 411
DoVerb 446
GetData 526
GetNativePointer 575
InsertClass 642
InsertFile 647
InsertObject 661

LinkTo 711
Open 769
Paste 818
PasteLink 820
PasteSpecial 822
ReleaseAutomationPointer 896
Save 928
SaveAs 934, 935
SelectObject 964
SetAutomationLocale 981
SetData 991
UpdateLinksDialog 1150

OLECustomControl functions
GetData 526
GetNativePointer 575
ReleaseAutomationPointer 896
SetAutomationLocale 981
SetData 991

OLEObject functions
ConnectToNewRemoteObject 392
ConnectToObject 393
ConnectToRemoteObject 396
DisconnectObject 443
GetAutomationNativePointer 507
ReleaseAutomationPointer 895
SetAutomationPointer 982
SetAutomationTimeout 983

OLEStorage functions
Clear 372
Close 378
MemberDelete 748
MemberExists 749
MemberRename 750
Open 769
SaveAs 936, 937

OLEStream functions
Close 379
Length 699
Open 769
Read 883
Seek 950
Write 1160

OPEN Cursor statement 160
Open event 276, 912
Open function 769
OpenChannel function 785
OpenSheet function 787
OpenSheetWithParm 789
OpenTab function 793
OpenTabWithParm function 796
OpenUserObject function 801
OpenUserObjectWithParm function 805
OpenWithParm 809
operating system
  information about 552
  RightToLeft version 672, 673, 674, 675, 678, 679, 914
operators
  about 69
  arithmetic 70
  assignment shortcuts 120, 122
  concatenation 73
  effect on datatypes 75
  logical 72
  precedence 74
  relational 72
OR operator 72
Original window 790
Other event 280
OutgoingCallList function 814
  oval
    and SetFocus function 1007
    printing 862
overflow on assignment 77
overlay 596, 1044
overloading functions 104
overriding functions 104

P
page
  printing 864
    printing borders 863, 865, 866
    size 847
PageCreated function 816
PageDown event 280
PageLeft event 281
PageRight event 283
PageUp event 284
paging functions
  ScrollNextPage 943
  ScrollPriorPage 945
paragraphs 1021
parameters
  command line 387
  opening sheets with 789
  opening tab pages with 796
  opening user objects with 794, 795, 802, 804, 805
  opening windows with 809
  specifying for DynamicDescriptionArea 1005
Parent pronoun 12
parent window
  changing position relative to 760
    obtaining 817
    of open window 770, 771, 809
parentheses in expressions 74
ParentWindow function 817
parsing strings 830, 832
password 733
Paste function 818
PasteLink function 820
PasteSpecial function 822
pasting
  embedding or linking 822
    from clipboard 818, 820
path
  of library file 700
  OLE storage 771
    returning 553
    saving files 558
pattern matching 743
PBDOTNET symbol 17
PBGetMenuString function 823
PBL file
  creating 700
    deleting 701
  listing contents of 701, 703
pbm_dwngraphcreate event 1040
PBNATIVE symbol 17
PBWEBFORM symbol 17
PBWEBSERVICE symbol 17
PBWINFORM symbol 17
PDB file 773
performance
  and Yield function 1169
  Any datatype 29
    dynamic function and event calls 100
Index

period in text patterns 744
Pi function 824
Picture functions
  ClassName 370
  Drag 447
  Draw 450
  Hide 616
  Move 760
  PointerX 826
  PointerY 827
  PostEvent 839
  Print 845
  SetFocus 1007
  SetPicture 1023
  SetPosition 1026
  SetRedraw 1032
  Show 1070
  TriggerEvent 1136
  TypeOf 1146
PictureListBox functions
  AddItem 335
  AddPicture 339
  DeletePicture 433
  DeletePictures 433
  FindItem 488
  InsertItem 650
  SelectedItem 953
  SelectItem 961
  SetTop 1061
  State 1092
  Text 1115
  Top 1122
  TotalItems 1124
  TotalSelected 1124
pictures
  for TreeView items 1012
  in listboxes 339
  in rich text 662
  in TreeView controls 339
  ListView controls 338, 341, 342
  overlay in lists 1020
  TreeView controls 342
PictureSelected event 285
pie graphs 531, 995
PIF file 925
PipeEnd event 285
Pipeline functions
  Cancel 359
  Repair 897
  Start 1082
PipeMeter event 286
PipeStart event 287
pixels 824, 1149
PixelsToUnits function 824
Play function 825
plus sign in text patterns 745
point size 856
pointer
determining distance from edge 826
distance from top 827
file 475, 477, 478, 480
read/write 950
returning object under 765
setting shape 1024
PointerX function 826
PointerY function 827
polymorphism for functions and events 98
PopupMenu function 828
PopulateError function 829
pop-up windows
  moving 760
  obtaining parent 817
  opening 771, 809
Pos function 830, 832
PosA function 832
position
  changing 760
  of insertion point 832
  setting for control 1026
Position function 832
positive numbers 1075
Post function 838
PostEvent function 839
posting functions or events 96
PostURL function 841
PowerBuilder units 824, 1149
PowerBuilder, datatypes for external functions 62
PowerObject base class 29, 81
PowerObject functions
  ClassName 370
  GetContextService 519
  GetParent 579
Index

PowerObjectParm
  and CloseWithReturn function 384
  determining type 1147
  opening sheets with parameters 790, 797, 799, 806, 808
PowerScript statements 120
preference of numeric datatypes 75
preference of operators 74
preprocessor symbols 17
presentation styles 1108
print cursor
  getting coordinates of 877
  in print jobs 847
Print function 845
print functions
Print 845
PrintBitmap 851
PrintCancel 852
PrintClose 854
PrintDataWindow 855
PrintDefineFont 856
PrintOpen 861
PrintOval 862
PrintPage 864
PrintRect 865
PrintRoundRect 866
PrintScreen 867
PrintSend 868
PrintSetFont 870
PrintSetSpacing 872
PrintSetup 873
PrintText 874
PrintWidth 876
PrintX 877
PrintY 877
print job 861
PrintBitmap function 851
PrintCancel function 852
PrintClose function 854
PrintDataWindow function 855
PrintDefineFont function 856
printer setup
  about 861
  and message boxes 752
PrintOval function 862
PrintPage function 864
PrintRect function 865
PrintRoundRect function 866
PrintScreen function 867
PrintSend function 868
PrintSetFont function 870
PrintSetPrinter function 871
PrintSetSpacing function 872
PrintSetup function 873
PrintSetupPrinter function 873
PrintText function 874
PrintWidth function 876
PrintX function 877
PrintY function 877
private access
  functions 60
  variables and constants 43
PRIVATEREAD access modifier 43
PRIVATEWRITE access modifier 44
processor 552
profile files
  reading 878, 880
  writing to 1028
ProfileClass objects, RoutineList function 924
ProfileInt function 878
ProfileLine objects, OutgoingCallList function 814
ProfileRoutine objects
  IncomingCallList function 629
  LineList function 710
  OutgoingCallList function 814
ProfileString function 880
Profiling functions
  BuildModel 355
  ClassList 368
  DestroyModel 437
  RoutineList 924
  SetTraceFileName 1061
  SystemRoutine 1112
ProgressIndex event 288
Index

pronouns
  about 11
  instance variables 36
  Parent 12
  Super 14
  This 13
properties
  and GetFocus function 562
  font, for printing 856
  getting and setting 505
  Message object 790
  setting width and height 908
  window 770, 773
property expressions, Any datatype 28
  PropertyChanged event 289
  PropertyRequestEdit event 290
protected access
  functions 60
  variables and constants 43
PROTECTEDREAD access modifier 43
PROTECTEDWRITE access modifier 44
public access
  functions 60
  variables and constants 43

Q
question mark
  dynamic SQL 172, 173, 176
  icon in message box 751
  in text patterns 745
quoted strings, continuing 15
quotes
  nesting 24
  rules for 25
  specifying 7
  with tilde 24

R
radians 823, 824
Rand function 881
random numbers
  initializing generator 882
  obtaining 881
Randomize function 882
RButtonDown event 291
RButtonUp event 293
Read function 883
read-only arguments 106
real datatype 23
Real function 886
recipient, mail 736
RecognizeText function 887
rectangle
  and SetFocus function 1007
  printing 865, 866
references
  and CloseWithReturn function 385
  passing arguments by 106
  passing parameters 791, 797, 799, 806, 808, 811, 813
  Registration database 643
  RegistryDelete function 887
  RegistryGet function 888
  RegistryKeys function 889
  RegistrySet function 891
  RegistryValues function 893
  relational operators 72
  RelativeDate function 893
  RelativeTime function 894
  ReleaseAutomationNativePointer function 895
  ReleaseNativePointer function 896
remainder 757
remote DDE application 911
remote procedure calls
  declaring 65
  defined 92
  RemoteExec event 294, 514, 1091
  RemoteHotLink event 294
  RemoteHotLinkStart event 1091
  RemoteHotLinkStop event 295, 1091
  RemoteRequest event 295, 992, 1091
  RemoteSend event 296, 529, 1091
  RemoveDirectory function 896
  Rename event 296
  Repair function 897
  repairing pipeline, canceling 359
  Replace function 898
  ReplaceA function 900
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReplaceText function</td>
<td>901</td>
</tr>
<tr>
<td>ReplaceW function</td>
<td>902</td>
</tr>
<tr>
<td>report view for ListView</td>
<td>567</td>
</tr>
<tr>
<td>reserved words</td>
<td>10</td>
</tr>
<tr>
<td>Reset function</td>
<td>902</td>
</tr>
<tr>
<td>ResetArgElements function</td>
<td>905</td>
</tr>
<tr>
<td>ResetDataColors function</td>
<td>906</td>
</tr>
<tr>
<td>ResetInk function</td>
<td>907</td>
</tr>
<tr>
<td>ResetPicture function</td>
<td>907</td>
</tr>
<tr>
<td>Resize event</td>
<td>297</td>
</tr>
<tr>
<td>Resize function</td>
<td>908</td>
</tr>
<tr>
<td>ResolveInitialReferences function</td>
<td>909</td>
</tr>
<tr>
<td>RespondRemote function</td>
<td>911</td>
</tr>
<tr>
<td>response windows</td>
<td></td>
</tr>
<tr>
<td>closing</td>
<td>383</td>
</tr>
<tr>
<td>moving</td>
<td>760</td>
</tr>
<tr>
<td>Restart function</td>
<td>912</td>
</tr>
<tr>
<td>ResumeTransaction function</td>
<td>912</td>
</tr>
<tr>
<td>retry button</td>
<td>751</td>
</tr>
<tr>
<td>RETURN statement</td>
<td>140</td>
</tr>
<tr>
<td>return values</td>
<td></td>
</tr>
<tr>
<td>about 109</td>
<td></td>
</tr>
<tr>
<td>event return codes</td>
<td>184</td>
</tr>
<tr>
<td>from ancestor events</td>
<td>115</td>
</tr>
<tr>
<td>from mail session</td>
<td>733</td>
</tr>
<tr>
<td>TriggerEvent function</td>
<td>1137</td>
</tr>
<tr>
<td>Reverse function</td>
<td>914</td>
</tr>
<tr>
<td>RevertToSelf function</td>
<td>915</td>
</tr>
<tr>
<td>RGB function</td>
<td>916</td>
</tr>
<tr>
<td>rich text</td>
<td></td>
</tr>
<tr>
<td>alignment</td>
<td>505, 979</td>
</tr>
<tr>
<td>and data</td>
<td>414</td>
</tr>
<tr>
<td>copying with formatting</td>
<td>402, 821</td>
</tr>
<tr>
<td>data</td>
<td>634, 635, 636, 637, 638</td>
</tr>
<tr>
<td>determining insertion point position</td>
<td>834</td>
</tr>
<tr>
<td>editing header and footer</td>
<td>1071</td>
</tr>
<tr>
<td>find again</td>
<td>496</td>
</tr>
<tr>
<td>finding text</td>
<td>483</td>
</tr>
<tr>
<td>formatting</td>
<td>579, 599, 604, 1021</td>
</tr>
<tr>
<td>line spacing</td>
<td>1048</td>
</tr>
<tr>
<td>preview</td>
<td>684</td>
</tr>
<tr>
<td>preview document</td>
<td>684, 843</td>
</tr>
<tr>
<td>printing</td>
<td>850, 857</td>
</tr>
<tr>
<td>save file</td>
<td>938</td>
</tr>
<tr>
<td>selecting</td>
<td>968</td>
</tr>
<tr>
<td>selecting a word</td>
<td>971</td>
</tr>
<tr>
<td>selecting all</td>
<td>970</td>
</tr>
<tr>
<td>text color</td>
<td>603, 1051</td>
</tr>
<tr>
<td>text settings</td>
<td>1051</td>
</tr>
<tr>
<td>RichTextEdit functions</td>
<td></td>
</tr>
<tr>
<td>CanUndo</td>
<td>360</td>
</tr>
<tr>
<td>Clear</td>
<td>372</td>
</tr>
<tr>
<td>Copy</td>
<td>401</td>
</tr>
<tr>
<td>CopyRTF</td>
<td>402</td>
</tr>
<tr>
<td>Cut</td>
<td>411</td>
</tr>
<tr>
<td>DataSource</td>
<td>414</td>
</tr>
<tr>
<td>Find</td>
<td>483</td>
</tr>
<tr>
<td>FindNext</td>
<td>496</td>
</tr>
<tr>
<td>GetAlignment</td>
<td>505</td>
</tr>
<tr>
<td>GetParagraphSetting</td>
<td>579</td>
</tr>
<tr>
<td>GetSpacing</td>
<td>599</td>
</tr>
<tr>
<td>GetTextColor</td>
<td>603</td>
</tr>
<tr>
<td>GetTextStyle</td>
<td>604</td>
</tr>
<tr>
<td>InputFieldChangeData</td>
<td>634</td>
</tr>
<tr>
<td>InputFieldCurrentName</td>
<td>635</td>
</tr>
<tr>
<td>InputFieldDeleteCurrent</td>
<td>636</td>
</tr>
<tr>
<td>InputFieldGetData</td>
<td>637</td>
</tr>
<tr>
<td>InputFieldInsert</td>
<td>638</td>
</tr>
<tr>
<td>InputFieldLocate</td>
<td>638</td>
</tr>
<tr>
<td>InsertPicture</td>
<td>662</td>
</tr>
<tr>
<td>IsPreview</td>
<td>684</td>
</tr>
<tr>
<td>LineCount</td>
<td>708</td>
</tr>
<tr>
<td>LineLength</td>
<td>709</td>
</tr>
<tr>
<td>Paste</td>
<td>818</td>
</tr>
<tr>
<td>PasteRTF</td>
<td>821</td>
</tr>
<tr>
<td>Position</td>
<td>834</td>
</tr>
<tr>
<td>Preview</td>
<td>843</td>
</tr>
<tr>
<td>Print</td>
<td>850</td>
</tr>
<tr>
<td>PrintEx</td>
<td>857</td>
</tr>
<tr>
<td>ReplaceText</td>
<td>901</td>
</tr>
<tr>
<td>SaveDocument</td>
<td>938</td>
</tr>
<tr>
<td>Scroll</td>
<td>942</td>
</tr>
<tr>
<td>ScrollNextPage</td>
<td>943, 944</td>
</tr>
<tr>
<td>ScrollPriorPage</td>
<td>945</td>
</tr>
<tr>
<td>ScrollPriorRow</td>
<td>946</td>
</tr>
<tr>
<td>ScrollToRow</td>
<td>947</td>
</tr>
<tr>
<td>SelectedColumn</td>
<td>952</td>
</tr>
<tr>
<td>SelectedLength</td>
<td>954</td>
</tr>
<tr>
<td>SelectedLine</td>
<td>956</td>
</tr>
<tr>
<td>SelectedPage</td>
<td>957</td>
</tr>
<tr>
<td>SelectedStart</td>
<td>958</td>
</tr>
</tbody>
</table>
Index

SelectedText 959
SelectText 968
SelectTextAll 970
SelectTextLine 971
SelectTextWord 971
SetAlignment 979
SetParagraphSetting 1021
SetSpacing 1048
SetTextColor 1051
SetTextStyle 1051
ShowHeadFoot 1071
Undo 1149
Right function 917
RightA 918
RightClicked event 298
RightDoubleClicked event 299
RightToLeft operating system 914
RightToLeft software 672, 673, 674, 675, 678, 679
RightTrim function 919, 920
RightW function 919
ROLLBACK statement 160
RollbackOnly function 920
RollbackTransaction function 921
Round function 923
RoutineList function 924
rows
  correcting pipeline data 897
  determining insertion point position 833
  scrolling 943, 944, 946, 947
rows, database
  deleting 155, 156
  fetching 158
  inserting 159
  updating 164
  updating cursored row 166
RPC see remote procedure calls
Run function 925
SaveDocument function 938
SaveInk function 940
SaveObject event 302
scatter graphs
  adding values to series 333
  changing data point values 758
  importing data 620, 623, 625, 627
  inserting data from strings 628
  obtaining data point values 524
scope operator 112
screen
  changing position relative to 760
  display 552
  distance to workspace 1158, 1159
  printing 867
scripts
  stopping execution 912
  terminating 140
  triggering events 1136
Scroll function 942
ScrollHorizontal event 752
scrolling
  ListBox 1061
  TreeView 1006
scrolling functions
  Scroll 942
  ScrollNextPage 943
  ScrollNextRow 944
  ScrollPriorPage 945
  ScrollPriorRow 946
  ScrollToRow 947
  Top 1122
ScrollNextPage function 943
ScrollNextRow function 944
ScrollPriorPage function 945
ScrollPriorRow function 946
ScrollToRow function 947
ScrollVertical event 752
searching, rich text 483, 496
Second function 948
SecondsAfter function 949
Seek function 950, 951
SeekType enumerated datatype 950
SELECT statement 161
SELECTBLOB statement 162
Selected event 303, 1018

S
Save As dialog box 933
Save event 301
Save File response window 558
Save function 928
Save function (InkPicture controls) 927

1200

PowerBuilder
SelectedColumn function 952
SelectedIndex function 953
SelectedItem function 953
SelectedLength function 954
SelectedLine function 956
SelectedPage function 957
SelectedStart function 958
SelectedText function 959
selection, clearing in list 962
SelectionChanged event 304
SelectionChanging event 307
SelectionRange function 960
SelectedItem function 961
SelectObject function 964
SelectText function 966
about 966
copying to clipboard 402
SelectTextAll function 970
SelectTextLine function 971
SelectTextWord function 971
Send function 973
sender 734
SendMessage function 973
series, graphs
adding to 340
adding values to 331, 351
clicked 765
counting 975
data points 413, 428, 524, 539, 757, 906
deleting 434, 903
finding number of 497
importing 620, 623, 627
inserting 663
inserting data 643
obtaining name 976
reporting appearance of 590
setting style 1039
SeriesCount function 975
SeriesName function 976
server application
activating 328, 964
closing DDE channel 383
connecting to 392, 393, 394, 396
DDE support 786
pasting and linking 820
providing data 584
sending data to 1032
sending to DDE client 992
stopping 1096
SetAbort function 977
SetAlignment function 979
SetArgElement function 980
SetAutomationPointer function 982
SetAutomationTimeout function 983
SetBoldDate function 985
SetByte function 986
SetComplete function 988
SetData function 991
SetDataDDE function 992
SetDataPieExplode function 995
SetDataStyle function 996
SetDataTransparency function 1002
SetDateLimits function 1004
SetDropHighlight function 1004
SetDynamicParm function 1005
SetFirstVisible function 1006
SetFocus function 1007
SetGlobalProperty function 1008
SetItem function 1009
SetLevelPictures function 1012
SetLibraryList function 1013
SetMask function 1015
SetMicroHelp function 1017
SetNewMobiLinkPassword function 1018
SetNull function 1019
SetOverlayPicture function 1020
SetParm function 1022
SetPicture function 1023
SetPointer function 1024
SetPosition function 1026
SetProfileString function 1028
SetRange function 1030
SetRecordSet function 1030
SetSelectedDate function 1036
SetSelectedRange function 1037
SetSeriesStyle function 1039
SetSeriesTransparency function 1046
SetState function 1049
SetSyncRegistryProperties function 1050
Index

SetTimeout function 1052
SetToday function 1054
SetToolbar function 1055
SetTop function 1061
SetTraceFileName function 1061
SetTransPool function 1063
setup printer 868
SetValue function 1063
shade
data points 533, 997
series 592, 1039
shapes
mouse pointer 1024
printing 863, 865, 866
shared objects
about 1067
SharedObjectDirectory function 1065
SharedObjectGet function 1065
SharedObjectRegister function 1069
SharedObjectUnregister function 1070
shared variables
about 34
initialized 42
SharedObjectDirectory function 1065
SharedObjectGet function 1065
SharedObjectRegister function 1069
SharedObjectUnregister function 1070
sharing data 414
sheets
arranging 345
getting active 504
getting first open 560
getting next open 576
obtaining parent 817
opening 771, 787, 789
toolbars 605, 607, 1055, 1057
Show event 309
Show function 1070
ShowHeadFoot function 1071
ShowHelp function 1072
ShowPopupHelp function 1074
Sign function 1075
SignalError function 1075
signing on to database during compile, preventing 148
Sin function 1076
sine 1076
SingleLineEdit functions
CanUndo 360
Clear 372
Copy 401
Cut 411
Move 760
Paste 818
Position 833
ReplaceText 901
SelectedLength 954
SelectedStart 958
SelectedText 959
SelectText 966
Undo 1149
size
changing 908
of screen 552
of string or blob 697, 699
Sleep function 1077
SNC, declaring procedure 170, 175, 180
solid fill pattern 1001, 1043
Sort event 310
Sort function 1077
sort order
and GetCalc function 614
when inserting items into lists 649
SortAll function 1079
sounds (beep) 349
source database 1082
Space function 1080
spaces
deleting leading 695, 696
deleting trailing 919, 920
inserting in a string 1080
removing from strings 1140, 1141
special ASCII characters in strings 6
SQL Native Client, declaring procedure 170
SQL statements
about 146
CLOSE Cursor 149
CLOSE Procedure 150
COMMIT 151
CONNECT 152
continuing 15
DECLARE Procedure 153
DISCONNECT 156
error handling 147
EXECUTE 157, 1005
FETCH 158
in pipeline execution 1083
INSERT 159
OPEN 1005
OPEN Cursor 160
painting 148
ROLLBACK 160
SELECT 161
SELECTBLOB 162
UPDATE 164
UPDATE Where Current of Cursor 166
UPDATEBLOB 165
SQLCode property 147
SQLDBCode property 147
SQLErrText property 147
SQLPreview event 312
Sqrt function 1081
square fill pattern 1001, 1043
square root 1081
Start event 313, 314
Start function 1082
about 1082
canceling pipeline 359
server application 393, 396
StartHotLink function 1089
StartServerDDE function 1091
storage
of listbox items 1092
setting highlighted 1049
State function 1092
statements, PowerScript
assignment 120
CALL 123
CHOOSE CASE 124
CONTINUE 126
CREATE 127
DESTROY 130
DO...LOOP 131
EXIT 133
FOR...NEXT 134
GOTO 136
HALT 137
IF...THEN 138
listed 119
RETURN 140
separating 16
static calls 98
StaticText control, inserting clipboard 374
stgShareMode enumerated datatype 777, 780
Stop function 1094
stop sign icon 751
StopHotLink function 1095
StopServerDDE function 1096
storages, OLE
file 934
releasing 378
saving 928
stored procedures
closing 150
declaring 148, 153
executing 157
stored procedures, declaring 65
streams, OLE
checking 749
deleting 748
renaming 750
string datatype 23
string function 1097
string functions
Asc 346, 347
Char 365
CharA 366
Fill 482
FillW 482, 483
Left 694, 695
LeftTrim 695, 696
LeftW 694, 695
Len 697, 699
Lower 725
Match 743
Mid 753
MidW 753
Pos 830, 832
Replace 898, 902
Right 917
RightTrim 919, 920
RightW 919
Space 1080
Trim 1140, 1141
Upper 1152
Index

StringParm property 790, 797, 799, 806, 808
strings
  char arrays 78
  comparing 72
  concatenating 73
  continuing 15
  converting 347, 351, 358, 417, 425, 445, 717, 719, 886
  converting to char 78
  deleting leading spaces 695, 696
  detecting contents 678, 683, 686
  determining width for printing 876
  extracting 365, 366, 753
  finding substrings 830, 832
  getting dynamic 550
  importing data from 627
  lowercase 725
  nested 24
  uppercase 1152
  writing to stream 1160
StringToObject function 1103
Stroke event 314
structure objects
  exporting as syntax 705
  listing 701
  recreating from syntax 706
structures
  about 79
  assignment 86
  autoinstantiated user objects 85
  for return values 384
  mailRecipient 737
  passing as arguments 107
  passing to external functions 64
  passing values as 811, 813
substorages, OLE
  checking 749
  deleting 748
  renaming 750
  saving 934
substrings
  extracting 753
  finding 830, 832
  replacing 898, 902
subtraction operator
  list of arithmetic operators 70
surrounded by spaces 16, 70
summary, moving objects to 1027
Super pronoun 14
SuspendTransaction function 1106
symbol types, graphs
  data points 535, 1000
  series 1043
Synchronize function 1107
syntax
  exporting object as 705
  recreating objects from 706
SyntaxFromSQL function 1108
system
  date 1121
  events 183, 838
  events, defined 92
  functions 112
  object classes 81
  object datatypes 29
  object hierarchy 29
  registry 887, 888, 889, 891, 893
  time 765
system and environment functions
  Clipboard 374
  CommandParm 387
  DebugBreak 425
  FindClassDefinition 486
  FindFunctionDefinition 487
  FindTypeDefinition 498
  GarbageCollect 502
  GarbageCollectGetTimeLimit 502
  GarbageCollectSetTimeLimit 503
  GetApplication 505
  GetEnvironment 552
  Handle 614
  PopulateError 829
  Post 838
  ProfileInt 878
  ProfileString 880
  Restart 912
  Run 925
  Send 973
  SetProfileString 1028
  ShowHelp 1072
  SignalError 1075
  Yield 1169
SystemError event 315
SystemKey event 316
SYSTEMREAD modifier 44
SystemRoutine function 1112
SYSTEMWRITE modifier 44

T

tab character, specifying 7
Tab functions
CloseTab 381
MoveTab 761
SelectTab 965
TabPostEvent 1113
TabTriggerEvent 1114
tab pages
changing order 761
CreatePage function 410
opening user objects 793, 796
PageCreated function 816
selecting 965
tables, database, transferring data between databases 1082
Tabular presentation style 1108
Tag property
and GetFocus function 562
storing MicroHelp text 1018
Tan function 1115
tangent 1115
target database for pipeline 1082
temporary files 734
terminator for string 354
text
deleting from edit controls 372
finding in RichTextEdit 483, 496
finding substrings 830, 832
importing data from string 627
line spacing when printing 847
metacharacters 744
MicroHelp 1017
obtaining current line 1115, 1116
of listbox item 953
of message box 751
on clipboard 374, 402, 411

pasting over 819
printing 846, 874
replacing 901
restoring 1149
save rich text as ASCII 938
selecting 954, 959, 966
setting color of 916
text file
importing data from 623
saving to 931
Text function 1115
Text property 562
TextLine function 1116
This pronoun 13
tilde
in strings 24
rules for 25
specifying 7
time
checking string 686
converting to datatype 1117
CPU 405
DateTime datatype 419
getting dynamic 546, 551
minutes 756
now 765
relative 894
seconds 948, 949
time datatype 25
Time function 1117
Timer event 317
Timer function 1120
timers, triggering event 1120
timing functions
CPU 405
Idle 618
Timer 1120
timing object
deactivating 1094
starting 1084
stopping 1094, 1095
title of message box 751
To Ansi function 1121
Today function 1121
ToolbarMoved event 319
toolbars 605, 607, 1055, 1057
Index

Moving objects to
Top function 1122

Topics
Calling Help 1072
Ending server application 1096
Starting server application 1091

TotalColumns function 1123
TotalItems function 1124
TotalSelected function 1124
ToUnicode 1125
ToUnicode function 1125
Trace file functions, Open 770
TraceBegin function 1126
TraceClose function 1127
TraceDisableActivity function 1128
TraceEnableActivity function 1130
TraceEnd function 1131
TraceError function 1132
TraceFile objects
Close function 379
NextActivity function 763
Reset function 904
TraceOpen function 1133
TraceTree objects
BuildModel function 355
DestroyModel function 437
EntryList function 454
SetTraceFileName function 1061
TraceTreeGarbageCollect objects, GetChildrenList function 511
TraceTreeObject objects, GetChildrenList function 511
TraceUser function 1135

Tracing functions
TraceBegin 1126
TraceClose 1127
TraceDisableActivity 1128
TraceEnableActivity 1130
TraceEnd 1131
TraceError 1132
TraceOpen 1133

TraceUser 1135
Trailer, moving objects to 1027
Transaction object functions
DBHandle 424
SyntaxFromSQL 1108
Transaction objects, creating 127
Transparent line style, graphs
Setting for data points 999
Setting for series 1042

TreeView functions
AddPicture 339
CollapseItem 386
DeleteItem 431
DeletePicture 433
DeletePictures 433
DeleteStatePicture 436
DeleteStatePictures 436
EditLabel 451
ExpandAll 459
ExpandItem 459
FindItem 492
GetItem 568
InsertItem 652, 653
InsertItemFirst 654
InsertItemLast 656
InsertItemSort 659
SelectItem 963
SetDropHighlight 1004
SetFirstVisible 1006
SetItem 1011
SetLevelPictures 1012
SetOverlayPicture 1020
Sort 1078
SortAll 1079

TrigEvent enumerated datatype 839
TriggerEvent function 1136

Triggering events 184
functions or events 96
TriggerPBEvent function 1138
Trim function 1140, 1141
Truncate function 1141
TrustVerify function 1143
TypeError function 1146
typographical conventions xxvi
Index

W

WaitForUploadAck event 323
warm link 455, 585, 785, 1034
WarningMessage event 323
week, day of 421, 422
Which function 1154
WHILE, in DO...LOOP statement 131
white space 16

width
  data point's line 998
  series line 1041
  setting 908
  string 876
  workspace 1158

Window ActiveX controls
  GetArgElement function 506
  GetLastReturn function 569
  InvokePBFunction function 668
  ResetArgElements function 905
  SetArgElement function 980
  TriggerPBEvent function 1138

Window functions
  ArrangeSheets 345
  ChangeMenu 364
  ClassName 370
  CloseUserObject 382
  Draw 450
  GetActiveSheet 504
  GetFirstSheet 560
  GetNextSheet 576
  Hide 616
  Move 760
  Open 769
  OpenSheet 787
  OpenSheetWithParm 789
  OpenTab 793
  OpenUserObject 801
  OpenWithParm 809
  ParentWindow 817
  PointerX 826
  PointerY 827
  PostEvent 839
  print 845
  Resize 908
  SetFocus 1007
  SetMicroHelp 1017
  SetPosition 1026
  SetRedraw 1032
  Show 1070
  TriggerEvent 1136
  TypeOf 1146
  WorkSpaceHeight 1156
  WorkSpaceWidth 1158
  WorkSpaceX 1158
  WorkSpaceY 1159

Window objects
  closing user objects 382
  exporting as syntax 705
  listing 701, 703
  recreating from syntax 706

Window painter 801, 803

windows
  adding user objects 793, 801, 805
  arranging 345, 787
  changing menus 364
  closing 376
  custom frames 1158, 1159
datatype of 769
  DDE conversation handle 1091
  getting active 504
  obtaining handle 614
  obtaining workspace height 1156
  obtaining workspace width 1158
  opening 769, 809
  posting messages 838
  setting position of 1026
  WordCap function 1156
  WordParm field
    and TriggerEvent function 1137
    posting events 839
workspace
  distance to screen 1158, 1159
  obtaining height of 1156
  obtaining width 1158

  WorkSpaceHeight function 1156
  WorkSpaceWidth function 1158
  WorkSpaceX function 1158
  WorkSpaceY function 1159
  Write function 1160
  Writes 1160
Index

X
x value
data point 524, 539, 597, 758, 1003, 1047
importing data 620, 623, 625, 627
inserting from strings 628
XMLParseFile function 1161
XMLParseString function 1165
xValue enumerated datatype 524, 539, 597, 1003, 1047

Y
y value
data point 524, 539, 597, 758, 1003, 1047
importing data 620, 623, 625, 627
inserting from strings 628
Year function 1168
year, about 418
Yield function 1169
yValue enumerated datatype 524, 539, 597, 1003, 1047

Z
zero, determining 1075
Index