Contents

About This Book ................................................................................................................ ....... xxiii

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 ............................................................................... 17

CHAPTER 2 Datatypes ....................................................................................... 19
Standard datatypes ................................................................. 19
The Any datatype ..................................................................... 24
System object datatypes ............................................................ 27
Enumerated datatypes ............................................................... 28
PowerBuilder datatypes in EAServer .............................................. 29

CHAPTER 3 Declarations................................................................................... 31
Declaring variables ................................................................. 31
  Where to declare variables ............................................... 32
  About using variables ....................................................... 33
  Syntax of a variable declaration ......................................... 35
Declaring constants ..................................................................... 45
# Contents

Declaring arrays ............................................................................. 46  
Values for array elements .......................................................... 49  
Size of variable-size arrays ....................................................... 50  
More about arrays ................................................................. 52  
Declaring external functions ........................................................... 55  
Datatypes for external function arguments .................................. 60  
Calling external functions ........................................................ 62  
Defining source for external functions ........................................... 62  
Declaring DBMS stored procedures as remote procedure calls .... 63

## CHAPTER 4 Operators and Expressions .......................................................... 67

Operators in PowerBuilder ............................................................. 67  
Arithmetic operators in PowerBuilder ........................................ 68  
Relational operators in PowerBuilder ........................................ 70  
Concatenation operator in PowerBuilder ..................................... 71  
Operator precedence in PowerBuilder expressions ...................... 72  
Datatype of PowerBuilder expressions ........................................ 73  
Numeric datatypes in PowerBuilder ........................................ 73  
String and char datatypes in PowerBuilder ............................. 76

## CHAPTER 5 Structures and Objects ................................................................. 77

About structures ............................................................................. 77  
About objects ................................................................................. 78  
About user objects.................................................................. 79  
Instantiating objects ............................................................... 81  
Using ancestors and descendants ............................................ 82  
Garbage collection ................................................................. 82  
User objects that behave like structures ................................. 83  
Assignment for objects and structures ........................................... 84  
Assignment for structures .......................................................... 84  
Assignment for objects ............................................................. 85  
Assignment for autoinstantiated user objects.......................... 86

## CHAPTER 6 Calling Functions and Events ...................................................... 89

About functions and events .......................................................... 89  
Finding and executing functions and events ............................... 92  
Finding functions ................................................................... 92  
Finding events ........................................................................ 93  
Triggering versus posting functions and events ......................... 94  
Static versus dynamic calls ...................................................... 96  
Static calls ........................................................................... 96  
Dynamic calls ....................................................................... 97
Contents

Overloading, overriding, and extending functions and events ..... 102
  Overloading and overriding functions ..................................... 102
  Extending and overriding events ............................................ 103
Passing arguments to functions and events ......................... 104
  Passing objects ................................................................... 104
  Passing structures ................................................................ 105
  Passing arrays .................................................................... 106
Using return values ................................................................. 107
  Functions ........................................................................... 107
  Events .................................................................................. 107
Using cascaded calling and return values ......................... 108
Syntax for calling PowerBuilder functions and events .......... 109
Calling functions and events in an object’s ancestor ............. 112

PART 2 STATEMENTS, EVENTS, AND FUNCTIONS

CHAPTER 7 PowerScript Statements............................................................. 117
  Assignment ............................................................................. 118
  CALL ..................................................................................... 121
  CHOOSE CASE ...................................................................... 122
  CONTINUE ............................................................................. 124
  CREATE ............................................................................... 125
  DESTROY .............................................................................. 128
  DO...LOOP............................................................................ 129
  EXIT ...................................................................................... 131
  FOR...NEXT ........................................................................ 132
  GOTO .................................................................................... 134
  HALT ..................................................................................... 135
  IF...THEN ............................................................................. 136
  RETURN ................................................................................ 138
  THROW ................................................................................ 139
  THROWS ............................................................................... 140
  TRY...CATCH...FINALLY...END TRY ........................................ 141

CHAPTER 8 SQL Statements .......................................................................... 143
  Using SQL in scripts................................................................ 144
    CLOSE rsor ......................................................................... 147
    CLOSE Procedure............................................................... 148
    COMMIT .............................................................................. 149
    CONNECT ............................................................................ 150
    DECLARE Cursor .............................................................. 151
    DECLARE Procedure ........................................................... 151
Contents

DELETE ................................................................. 153
DELETE Where Current of Cursor .............................. 154
DISCONNECT ......................................................... 154
EXECUTE ............................................................... 155
FETCH ................................................................. 156
INSERT ................................................................. 157
OPEN Cursor ........................................................... 158
ROLLBACK ............................................................... 158
SELECT ................................................................. 159
SELECTBLOB ............................................................ 160
UPDATE ................................................................. 162
UPDATEBLOB ........................................................... 163
UPDATE Where Current of Cursor .............................. 164

Using dynamic SQL ....................................................... 165
Dynamic SQL Format 1 .................................................. 168
Dynamic SQL Format 2 .................................................. 169
Dynamic SQL Format 3 .................................................. 170
Dynamic SQL Format 4 .................................................. 173

CHAPTER 9 PowerScript Events ........................................ 179
About events ............................................................... 179
Activate ................................................................. 182
BeginDrag ............................................................... 183
BeginLabelEdit .......................................................... 186
BeginRightDrag .......................................................... 187
Clicked ................................................................. 190
Close ................................................................. 196
CloseQuery ............................................................. 198
CloseUp ............................................................... 199
ColumnClick ........................................................... 200
Constructor .............................................................. 201
DataChange .............................................................. 202
DateChanged ............................................................ 202
DateSelected ............................................................ 203
Deactivate .............................................................. 204
DeleteAllItems .......................................................... 204
DeleteItem ............................................................. 205
Destructor ............................................................... 206
DoubleClick ............................................................ 207
DragDrop ............................................................... 210
DragEnter .............................................................. 214
DragLeave ............................................................. 215
DragWithin ............................................................. 217
DropDown .............................................................. 219
Contents

EndLabelEdit................................................................................ 220
Error ............................................................................................. 222
ExternalException ........................................................................ 225
FileExists...................................................................................... 228
Gesture ........................................................................................ 229
GetFocus...................................................................................... 230
Help.............................................................................................. 232
Hide .............................................................................................. 233
HotLinkAlarm................................................................................ 233
Idle ............................................................................................... 234
InputFieldSelected ....................................................................... 234
InsertItem ..................................................................................... 235
ItemActivate .................................................................................. 235
ItemChanged ................................................................................ 236
ItemChanging ............................................................................. 237
ItemCollapsed .............................................................................. 238
ItemCollapsing ............................................................................. 239
ItemExpanded .............................................................................. 240
ItemExpanding ............................................................................. 241
ItemPopulate ................................................................................ 242
Key ............................................................................................... 243
LineDown ..................................................................................... 244
LineLeft ........................................................................................ 245
LineRight ...................................................................................... 246
LineUp .......................................................................................... 247
LoseFocus .................................................................................... 248
Modified ........................................................................................ 250
MouseDown ................................................................................. 252
MouseMove.................................................................................. 254
MouseUp...................................................................................... 257
Moved .......................................................................................... 260
Notify ............................................................................................ 260
Open ............................................................................................ 261
Other ............................................................................................ 264
PageDown.................................................................................... 265
PageLeft ....................................................................................... 266
PageRight .................................................................................... 267
PageUp ........................................................................................ 268
PictureSelected ............................................................................ 269
PipeEnd ....................................................................................... 269
PipeMeter ..................................................................................... 270
PipeStart ..................................................................................... 271
PrintFooter ................................................................................... 271
PrintHeader .................................................................................. 272
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddStatePicture</td>
<td>322</td>
</tr>
<tr>
<td>AddToLibraryList</td>
<td>323</td>
</tr>
<tr>
<td>Arrange</td>
<td>324</td>
</tr>
<tr>
<td>ArrangeSheets</td>
<td>325</td>
</tr>
<tr>
<td>Asc</td>
<td>326</td>
</tr>
<tr>
<td>AscA</td>
<td>326</td>
</tr>
<tr>
<td>ASin</td>
<td>328</td>
</tr>
<tr>
<td>ATan</td>
<td>329</td>
</tr>
<tr>
<td>Beep</td>
<td>329</td>
</tr>
<tr>
<td>BeginTransaction</td>
<td>330</td>
</tr>
<tr>
<td>Blob</td>
<td>331</td>
</tr>
<tr>
<td>BlobEdit</td>
<td>332</td>
</tr>
<tr>
<td>BlobMid</td>
<td>334</td>
</tr>
<tr>
<td>BuildModel</td>
<td>335</td>
</tr>
<tr>
<td>Byte</td>
<td>338</td>
</tr>
<tr>
<td>Cancel</td>
<td>339</td>
</tr>
<tr>
<td>CanUndo</td>
<td>339</td>
</tr>
<tr>
<td>CategoryCount</td>
<td>340</td>
</tr>
<tr>
<td>CategoryName</td>
<td>341</td>
</tr>
<tr>
<td>Ceiling</td>
<td>342</td>
</tr>
<tr>
<td>ChangeDirectory</td>
<td>342</td>
</tr>
<tr>
<td>ChangeMenu</td>
<td>343</td>
</tr>
<tr>
<td>Char</td>
<td>344</td>
</tr>
<tr>
<td>CharA</td>
<td>345</td>
</tr>
<tr>
<td>Check</td>
<td>346</td>
</tr>
<tr>
<td>ChooseColor</td>
<td>347</td>
</tr>
<tr>
<td>ClassList</td>
<td>347</td>
</tr>
<tr>
<td>ClassName</td>
<td>349</td>
</tr>
<tr>
<td>Clear</td>
<td>351</td>
</tr>
<tr>
<td>ClearBoldDates</td>
<td>352</td>
</tr>
<tr>
<td>Clipboard</td>
<td>353</td>
</tr>
<tr>
<td>Close</td>
<td>355</td>
</tr>
<tr>
<td>CloseChannel</td>
<td>359</td>
</tr>
<tr>
<td>CloseTab</td>
<td>360</td>
</tr>
<tr>
<td>CloseUserObject</td>
<td>361</td>
</tr>
<tr>
<td>CloseWithReturn</td>
<td>362</td>
</tr>
<tr>
<td>CollapseItem</td>
<td>365</td>
</tr>
<tr>
<td>CommandParm</td>
<td>366</td>
</tr>
<tr>
<td>CommitTransaction</td>
<td>367</td>
</tr>
<tr>
<td>ConnectToNewObject</td>
<td>369</td>
</tr>
<tr>
<td>ConnectToNewRemoteObject</td>
<td>371</td>
</tr>
<tr>
<td>ConnectToObject</td>
<td>372</td>
</tr>
<tr>
<td>ConnectToRemoteObject</td>
<td>375</td>
</tr>
<tr>
<td>ConnectToServer</td>
<td>377</td>
</tr>
</tbody>
</table>
Copy ................................................................. 380
CopyRTF ......................................................... 381
Cos ................................................................. 383
Cpu ................................................................. 384
CreateDirectory ............................................ 384
CreateInstance ............................................. 385
CreatePage .................................................... 389
Cut ................................................................. 390
DataCount ..................................................... 392
DataSource ................................................... 393
Date ............................................................... 395
DateTime ....................................................... 398
day .............................................................. 400
DayName ....................................................... 401
DayNumber ................................................... 402
DaysAfter ..................................................... 403
DBHandle ...................................................... 404
DebugBreak .................................................. 405
Dec ................................................................. 405
DeleteCategory .............................................. 406
DeleteColumn ............................................... 407
DeleteColumns .............................................. 407
DeleteData .................................................... 408
DeleteItem ..................................................... 408
DeleteItems .................................................... 411
DeleteLargePicture ....................................... 412
DeleteLargePictures ...................................... 412
DeletePicture ............................................... 413
DeletePictures .............................................. 413
DeleteSeries ................................................ 414
DeleteSmallPicture ...................................... 415
DeleteSmallPictures ..................................... 415
DeleteStatePicture ....................................... 416
DeleteStatePictures ..................................... 416
DestroyModel ............................................... 417
DirectoryExists .......................................... 417
DirList ........................................................ 418
DirSelect ...................................................... 420
Disable ......................................................... 421
DisableCommit ............................................ 422
DisconnectObject ........................................ 423
DisconnectServer ....................................... 424
Double ......................................................... 425
DoVerb ........................................................ 426
Drag ............................................................................................. 427
DraggedObject ............................................................................. 428
Draw ............................................................................................. 430
EditLabel ...................................................................................... 431
Enable .......................................................................................... 432
EnableCommit ............................................................................. 433
EntryList ....................................................................................... 434
ExecRemote ................................................................................... 435
Exp ............................................................................................... 438
ExpandAll ..................................................................................... 439
ExpandItem ................................................................................... 439
Fact .............................................................................................. 440
FileClose ...................................................................................... 441
FileCopy ....................................................................................... 441
FileDelete ................................................................................... 442
FileEncoding ................................................................................ 443
FileExists .................................................................................... 444
FileLength .................................................................................... 445
FileLength64 ................................................................................ 446
FileMove ....................................................................................... 447
FileOpen ....................................................................................... 448
FileRead ....................................................................................... 452
FileReadEx .................................................................................. 454
FileSeek ....................................................................................... 456
FileSeek64 .................................................................................. 457
FileWrite ....................................................................................... 459
FileWriteEx .................................................................................. 461
Fill ................................................................................................. 463
FillA .............................................................................................. 464
FillW ............................................................................................. 464
Find .............................................................................................. 464
FindCategory ............................................................................... 466
FindClassDefinition ..................................................................... 467
FindFunctionDefinition ................................................................. 468
FindItem ....................................................................................... 469
FindMatchingFunction ................................................................. 475
FindNext ....................................................................................... 477
FindSeries ................................................................................... 478
FindTypeDefinition ....................................................................... 479
FromAnsi ...................................................................................... 481
FromUnicode ............................................................................... 481
GarbageCollect ........................................................................... 482
GarbageCollectGetTimeLimit ....................................................... 482
GarbageCollectSetTimeLimit ....................................................... 483
Contents

GetActiveSheet ................................................................. 484
GetAlignment ................................................................. 485
GetApplication ............................................................... 485
GetArgElement ............................................................... 486
GetAutomationNativePointer .......................................... 487
GetByte ....................................................................... 488
GetCertificateLabel ....................................................... 489
GetChildrenList .......................................................... 491
GetColumn ................................................................. 493
GetCommandDDE ............................................................ 494
GetCommandDDEOrigin .................................................. 495
GetCompanyName ........................................................... 496
GetContextKeywords ...................................................... 497
GetContextService ......................................................... 498
GetCredentialAttribute ................................................. 500
GetCurrentDirectory ....................................................... 502
GetData ................................................................. 502
GetDataDDE ................................................................. 507
GetDataDDEOrigin ......................................................... 507
GetDataPieExplode ....................................................... 509
GetDataStyle ............................................................... 510
GetDataValue ............................................................. 516
GetDateLimits ............................................................... 518
GetDisplayRange .......................................................... 519
GetDynamicDate ........................................................... 520
GetDynamicDateTime .................................................... 522
GetDynamicNumber ....................................................... 524
GetDynamicString ........................................................ 525
GetDynamicTime ........................................................ 526
GetEnvironment .......................................................... 527
GetFileOpenName ........................................................ 528
GetFileSaveName ......................................................... 533
GetFirstSheet .............................................................. 535
GetFixesVersion .......................................................... 536
GetFocus ................................................................. 537
GetFolder ................................................................. 538
GetGlobalProperty ....................................................... 539
GetHostObject ............................................................. 540
GetItem ................................................................. 541
GetItemAtPointer ........................................................ 544
GetLastReturn ............................................................. 545
GetLibraryList ............................................................. 546
GetMajorVersion .......................................................... 547
GetMessage ............................................................... 548
<table>
<thead>
<tr>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetMinorVersion</td>
<td>549</td>
</tr>
<tr>
<td>GetName</td>
<td>550</td>
</tr>
<tr>
<td>GetNativePointer</td>
<td>551</td>
</tr>
<tr>
<td>getNextSheet</td>
<td>552</td>
</tr>
<tr>
<td>GetOrigin</td>
<td>553</td>
</tr>
<tr>
<td>GetParagraphSetting</td>
<td>554</td>
</tr>
<tr>
<td>GetParent</td>
<td>554</td>
</tr>
<tr>
<td>GetPin</td>
<td>556</td>
</tr>
<tr>
<td>GetRecordSet</td>
<td>558</td>
</tr>
<tr>
<td>GetRemote</td>
<td>559</td>
</tr>
<tr>
<td>GetSelectedDate</td>
<td>562</td>
</tr>
<tr>
<td>GetSelectedRange</td>
<td>563</td>
</tr>
<tr>
<td>GetSeriesStyle</td>
<td>564</td>
</tr>
<tr>
<td>GetShortName</td>
<td>570</td>
</tr>
<tr>
<td>GetSpacing</td>
<td>571</td>
</tr>
<tr>
<td>GetStatus</td>
<td>571</td>
</tr>
<tr>
<td>GetText</td>
<td>573</td>
</tr>
<tr>
<td>GetTextColor</td>
<td>574</td>
</tr>
<tr>
<td>GetTextStyle</td>
<td>574</td>
</tr>
<tr>
<td>GetToday</td>
<td>575</td>
</tr>
<tr>
<td>GetToolbar</td>
<td>576</td>
</tr>
<tr>
<td>GetToolbarPos</td>
<td>578</td>
</tr>
<tr>
<td>GetTransactionName</td>
<td>581</td>
</tr>
<tr>
<td>GetURL</td>
<td>582</td>
</tr>
<tr>
<td>GetValue</td>
<td>583</td>
</tr>
<tr>
<td>GetVersionName</td>
<td>584</td>
</tr>
<tr>
<td>Handle</td>
<td>585</td>
</tr>
<tr>
<td>Hide</td>
<td>587</td>
</tr>
<tr>
<td>Hour</td>
<td>588</td>
</tr>
<tr>
<td>HyperLinkToURL</td>
<td>588</td>
</tr>
<tr>
<td>Idle</td>
<td>589</td>
</tr>
<tr>
<td>ImpersonateClient</td>
<td>590</td>
</tr>
<tr>
<td>ImportClipboard</td>
<td>591</td>
</tr>
<tr>
<td>ImportFile</td>
<td>594</td>
</tr>
<tr>
<td>ImportString</td>
<td>598</td>
</tr>
<tr>
<td>IncomingCallList</td>
<td>600</td>
</tr>
<tr>
<td>Init</td>
<td>602</td>
</tr>
<tr>
<td>InputFieldChangeData</td>
<td>605</td>
</tr>
<tr>
<td>InputFieldCurrentName</td>
<td>607</td>
</tr>
<tr>
<td>InputFieldDeleteCurrent</td>
<td>608</td>
</tr>
<tr>
<td>InputFieldGetData</td>
<td>609</td>
</tr>
<tr>
<td>InputFieldInsert</td>
<td>609</td>
</tr>
<tr>
<td>InputFieldLocate</td>
<td>610</td>
</tr>
<tr>
<td>InsertCategory</td>
<td>612</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------</td>
</tr>
<tr>
<td>InsertClass</td>
<td>613</td>
</tr>
<tr>
<td>InsertColumn</td>
<td>614</td>
</tr>
<tr>
<td>InsertData</td>
<td>615</td>
</tr>
<tr>
<td>InsertDocument</td>
<td>617</td>
</tr>
<tr>
<td>InsertFile</td>
<td>619</td>
</tr>
<tr>
<td>InsertItem</td>
<td>620</td>
</tr>
<tr>
<td>InsertItemFirst</td>
<td>627</td>
</tr>
<tr>
<td>InsertItemLast</td>
<td>629</td>
</tr>
<tr>
<td>InsertItemSort</td>
<td>631</td>
</tr>
<tr>
<td>InsertObject</td>
<td>633</td>
</tr>
<tr>
<td>InsertPicture</td>
<td>634</td>
</tr>
<tr>
<td>InsertSeries</td>
<td>635</td>
</tr>
<tr>
<td>Int</td>
<td>636</td>
</tr>
<tr>
<td>Integer</td>
<td>636</td>
</tr>
<tr>
<td>InternetData</td>
<td>638</td>
</tr>
<tr>
<td>IntHigh</td>
<td>639</td>
</tr>
<tr>
<td>IntLow</td>
<td>639</td>
</tr>
<tr>
<td>IntLow</td>
<td>639</td>
</tr>
<tr>
<td>InvokePBFunction</td>
<td>640</td>
</tr>
<tr>
<td>_Is_A</td>
<td>642</td>
</tr>
<tr>
<td>IsAlive</td>
<td>643</td>
</tr>
<tr>
<td>IsAllArabic</td>
<td>644</td>
</tr>
<tr>
<td>IsAllHebrew</td>
<td>644</td>
</tr>
<tr>
<td>IsAnyArabic</td>
<td>645</td>
</tr>
<tr>
<td>IsAnyHebrew</td>
<td>646</td>
</tr>
<tr>
<td>IsArabic</td>
<td>646</td>
</tr>
<tr>
<td>IsArabicAndNumbers</td>
<td>647</td>
</tr>
<tr>
<td>IsCallerInRole</td>
<td>648</td>
</tr>
<tr>
<td>IsDate</td>
<td>650</td>
</tr>
<tr>
<td>IsHebrew</td>
<td>650</td>
</tr>
<tr>
<td>IsHebrewAndNumbers</td>
<td>651</td>
</tr>
<tr>
<td>IsImpersonating</td>
<td>652</td>
</tr>
<tr>
<td>IsInTransaction</td>
<td>653</td>
</tr>
<tr>
<td>IsNull</td>
<td>654</td>
</tr>
<tr>
<td>IsNumber</td>
<td>655</td>
</tr>
<tr>
<td>IsPreview</td>
<td>656</td>
</tr>
<tr>
<td>IsSecurityEnabled</td>
<td>656</td>
</tr>
<tr>
<td>IsTime</td>
<td>658</td>
</tr>
<tr>
<td>IsTransactionAborted</td>
<td>659</td>
</tr>
<tr>
<td>IsValid</td>
<td>660</td>
</tr>
<tr>
<td>KeyDown</td>
<td>661</td>
</tr>
<tr>
<td>LastPos</td>
<td>664</td>
</tr>
<tr>
<td>Left</td>
<td>666</td>
</tr>
<tr>
<td>LeftA</td>
<td>667</td>
</tr>
<tr>
<td>LeftW</td>
<td>667</td>
</tr>
</tbody>
</table>
Contents

LeftTrim ................................................................. 667
LeftTrimW ............................................................ 668
Len ................................................................. 668
LenA ............................................................... 670
LenW ............................................................... 670
Length ............................................................. 671
LibraryCreate .................................................... 672
LibraryDelete .................................................... 673
LibraryDirectory ............................................... 673
LibraryDirectoryEx ............................................. 675
LibraryExport ................................................... 677
LibraryImport ................................................... 678
LineCount ........................................................ 679
LineLength ....................................................... 680
LineList .......................................................... 681
LinkTo ............................................................ 683
LoadInk ........................................................... 684
LoadPicture ...................................................... 685
Log ................................................................. 686
LogTen ............................................................. 687
Long ............................................................... 688
LongLong ......................................................... 690
Lookup ........................................................... 692
Lower ............................................................. 697
LowerBound ....................................................... 697
mailAddress ..................................................... 699
mailDeleteMessage .......................................... 701
mailGetMessages ............................................. 702
mailHandle ...................................................... 703
mailLogoff ....................................................... 704
mailLogon ....................................................... 705
mailReadMessage ............................................ 706
mailRecipientDetails ....................................... 709
mailResolveRecipient .................................... 710
mailSaveMessage ............................................ 712
mailSend ........................................................ 715
Match ............................................................ 716
MatchW .......................................................... 719
Max ............................................................... 720
MemberDelete .................................................. 721
MemberExists .................................................. 722
MemberRename ............................................... 723
MessageBox ...................................................... 724
Mid ............................................................... 726
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostEvent</td>
<td>811</td>
</tr>
<tr>
<td>PostURL</td>
<td>814</td>
</tr>
<tr>
<td>Preview</td>
<td>816</td>
</tr>
<tr>
<td>Print</td>
<td>817</td>
</tr>
<tr>
<td>PrintBitmap</td>
<td>823</td>
</tr>
<tr>
<td>PrintCancel</td>
<td>824</td>
</tr>
<tr>
<td>PrintClose</td>
<td>826</td>
</tr>
<tr>
<td>PrintDataWindow</td>
<td>827</td>
</tr>
<tr>
<td>PrintDefineFont</td>
<td>828</td>
</tr>
<tr>
<td>PrintEx</td>
<td>829</td>
</tr>
<tr>
<td>PrintGetPrinter</td>
<td>830</td>
</tr>
<tr>
<td>PrintGetPrinters</td>
<td>831</td>
</tr>
<tr>
<td>PrintLine</td>
<td>832</td>
</tr>
<tr>
<td>PrintOpen</td>
<td>833</td>
</tr>
<tr>
<td>PrintOval</td>
<td>834</td>
</tr>
<tr>
<td>PrintPage</td>
<td>836</td>
</tr>
<tr>
<td>PrintRect</td>
<td>837</td>
</tr>
<tr>
<td>PrintRoundRect</td>
<td>838</td>
</tr>
<tr>
<td>PrintScreen</td>
<td>839</td>
</tr>
<tr>
<td>PrintSend</td>
<td>840</td>
</tr>
<tr>
<td>PrintSetFont</td>
<td>842</td>
</tr>
<tr>
<td>PrintSetPrinter</td>
<td>843</td>
</tr>
<tr>
<td>PrintSetSpacing</td>
<td>844</td>
</tr>
<tr>
<td>PrintSetup</td>
<td>845</td>
</tr>
<tr>
<td>PrintSetupPrinter</td>
<td>845</td>
</tr>
<tr>
<td>PrintText</td>
<td>846</td>
</tr>
<tr>
<td>PrintWidth</td>
<td>848</td>
</tr>
<tr>
<td>PrintX</td>
<td>849</td>
</tr>
<tr>
<td>PrintY</td>
<td>849</td>
</tr>
<tr>
<td>ProfileInt</td>
<td>850</td>
</tr>
<tr>
<td>ProfileString</td>
<td>852</td>
</tr>
<tr>
<td>Rand</td>
<td>854</td>
</tr>
<tr>
<td>Randomize</td>
<td>854</td>
</tr>
<tr>
<td>Read</td>
<td>855</td>
</tr>
<tr>
<td>Real</td>
<td>858</td>
</tr>
<tr>
<td>RecognizeText</td>
<td>859</td>
</tr>
<tr>
<td>RegistryDelete</td>
<td>859</td>
</tr>
<tr>
<td>RegistryGet</td>
<td>860</td>
</tr>
<tr>
<td>RegistryKeys</td>
<td>861</td>
</tr>
<tr>
<td>RegistrySet</td>
<td>862</td>
</tr>
<tr>
<td>RegistryValues</td>
<td>865</td>
</tr>
<tr>
<td>RelativeDate</td>
<td>866</td>
</tr>
<tr>
<td>RelativeTime</td>
<td>866</td>
</tr>
<tr>
<td>ReleaseAutomationNativePointer</td>
<td>867</td>
</tr>
</tbody>
</table>
Contents

ReleaseNativePointer .......................................................... 868
RemoveDirectory ................................................................. 868
Repair .................................................................................. 869
Replace .................................................................................. 870
ReplaceA ............................................................................... 872
ReplaceText .......................................................................... 873
ReplaceW .............................................................................. 874
Reset .................................................................................... 874
ResetArgElements ................................................................ 877
ResetDataColors .................................................................. 878
ResetInk .............................................................................. 879
ResetPicture ......................................................................... 879
Resize ................................................................................... 880
Resolve_Initial_References ..................................................... 881
RespondRemote ...................................................................... 883
Restart ................................................................................... 884
ResumeTransaction ............................................................... 884
Reverse .................................................................................. 886
RevertToSelf .......................................................................... 887
RGB ..................................................................................... 887
Right ...................................................................................... 889
RightA ................................................................................... 890
RightW .................................................................................. 890
RightTrim .............................................................................. 890
RightTrimW .......................................................................... 891
RollbackOnly ......................................................................... 891
RollbackTransaction .............................................................. 893
Round .................................................................................... 894
RoutineList ............................................................................ 895
Run ....................................................................................... 896
Save ...................................................................................... 898
SaveAs ................................................................................... 902
SaveDocument ....................................................................... 909
SaveInk ................................................................................ 911
Scroll ..................................................................................... 914
ScrollNextPage ...................................................................... 915
ScrollNextRow ...................................................................... 916
ScrollPriorPage ..................................................................... 917
ScrollPriorRow ..................................................................... 918
ScrollToRow .......................................................................... 919
Second .................................................................................. 920
SecondsAfter ......................................................................... 920
Seek ..................................................................................... 921
SelectedColumn ..................................................................... 923
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectedIndex</td>
<td>924</td>
</tr>
<tr>
<td>SelectedItem</td>
<td>925</td>
</tr>
<tr>
<td>SelectedLength</td>
<td>926</td>
</tr>
<tr>
<td>SelectedLine</td>
<td>927</td>
</tr>
<tr>
<td>SelectedPage</td>
<td>928</td>
</tr>
<tr>
<td>SelectedStart</td>
<td>929</td>
</tr>
<tr>
<td>SelectedText</td>
<td>930</td>
</tr>
<tr>
<td>SelectionRange</td>
<td>931</td>
</tr>
<tr>
<td>SelectItem</td>
<td>932</td>
</tr>
<tr>
<td>SelectObject</td>
<td>936</td>
</tr>
<tr>
<td>SelectTab</td>
<td>936</td>
</tr>
<tr>
<td>SelectText</td>
<td>937</td>
</tr>
<tr>
<td>SelectTextAll</td>
<td>941</td>
</tr>
<tr>
<td>SelectTextLine</td>
<td>942</td>
</tr>
<tr>
<td>SelectTextWord</td>
<td>943</td>
</tr>
<tr>
<td>Send</td>
<td>945</td>
</tr>
<tr>
<td>SeriesCount</td>
<td>947</td>
</tr>
<tr>
<td>SeriesName</td>
<td>948</td>
</tr>
<tr>
<td>SetAbort</td>
<td>949</td>
</tr>
<tr>
<td>SetAlignment</td>
<td>951</td>
</tr>
<tr>
<td>SetArgElement</td>
<td>952</td>
</tr>
<tr>
<td>SetAutomationLocale</td>
<td>953</td>
</tr>
<tr>
<td>SetAutomationPointer</td>
<td>954</td>
</tr>
<tr>
<td>SetAutomationTimeout</td>
<td>955</td>
</tr>
<tr>
<td>SetBoldDate</td>
<td>957</td>
</tr>
<tr>
<td>SetByte</td>
<td>958</td>
</tr>
<tr>
<td>SetColumn</td>
<td>959</td>
</tr>
<tr>
<td>SetComplete</td>
<td>960</td>
</tr>
<tr>
<td>SetData</td>
<td>963</td>
</tr>
<tr>
<td>SetDataDDE</td>
<td>964</td>
</tr>
<tr>
<td>SetDataPieExplode</td>
<td>965</td>
</tr>
<tr>
<td>SetDataStyle</td>
<td>967</td>
</tr>
<tr>
<td>SetDateLimits</td>
<td>972</td>
</tr>
<tr>
<td>SetDropHighlight</td>
<td>973</td>
</tr>
<tr>
<td>SetDynamicParm</td>
<td>974</td>
</tr>
<tr>
<td>SetFirstVisible</td>
<td>975</td>
</tr>
<tr>
<td>SetFocus</td>
<td>976</td>
</tr>
<tr>
<td>SetGlobalProperty</td>
<td>977</td>
</tr>
<tr>
<td>SetItem</td>
<td>978</td>
</tr>
<tr>
<td>SetLevelPictures</td>
<td>981</td>
</tr>
<tr>
<td>SetLibraryList</td>
<td>982</td>
</tr>
<tr>
<td>SetMask</td>
<td>984</td>
</tr>
<tr>
<td>SetMessage</td>
<td>985</td>
</tr>
<tr>
<td>SetMicroHelp</td>
<td>986</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SetNull</td>
<td>987</td>
</tr>
<tr>
<td>SetOverlayPicture</td>
<td>988</td>
</tr>
<tr>
<td>SetParagraphSetting</td>
<td>990</td>
</tr>
<tr>
<td>SetPicture</td>
<td>991</td>
</tr>
<tr>
<td>SetPointer</td>
<td>992</td>
</tr>
<tr>
<td>SetPosition</td>
<td>993</td>
</tr>
<tr>
<td>SetProfileString</td>
<td>995</td>
</tr>
<tr>
<td>SetRange</td>
<td>997</td>
</tr>
<tr>
<td>SetRecordSet</td>
<td>998</td>
</tr>
<tr>
<td>SetRedraw</td>
<td>999</td>
</tr>
<tr>
<td>SetRemote</td>
<td>1000</td>
</tr>
<tr>
<td>SetResultSet</td>
<td>1002</td>
</tr>
<tr>
<td>SetSeriesStyle</td>
<td>1003</td>
</tr>
<tr>
<td>SetSelectedDate</td>
<td>1010</td>
</tr>
<tr>
<td>SetSelectedRange</td>
<td>1011</td>
</tr>
<tr>
<td>SetSpacing</td>
<td>1012</td>
</tr>
<tr>
<td>SetState</td>
<td>1013</td>
</tr>
<tr>
<td>SetTextColor</td>
<td>1014</td>
</tr>
<tr>
<td>SetTextColor</td>
<td>1014</td>
</tr>
<tr>
<td>SetTextStyle</td>
<td>1014</td>
</tr>
<tr>
<td>SetTimeout</td>
<td>1015</td>
</tr>
<tr>
<td>SetToday</td>
<td>1017</td>
</tr>
<tr>
<td>SetToolbar</td>
<td>1018</td>
</tr>
<tr>
<td>SetToolbarPos</td>
<td>1020</td>
</tr>
<tr>
<td>SetTop</td>
<td>1024</td>
</tr>
<tr>
<td>SetTraceFileName</td>
<td>1024</td>
</tr>
<tr>
<td>SetTransPool</td>
<td>1026</td>
</tr>
<tr>
<td>SetValue</td>
<td>1028</td>
</tr>
<tr>
<td>SharedObjectDirectory</td>
<td>1029</td>
</tr>
<tr>
<td>SharedObjectGet</td>
<td>1030</td>
</tr>
<tr>
<td>SharedObjectRegister</td>
<td>1033</td>
</tr>
<tr>
<td>SharedObjectUnregister</td>
<td>1034</td>
</tr>
<tr>
<td>Show</td>
<td>1034</td>
</tr>
<tr>
<td>ShowHeadFoot</td>
<td>1035</td>
</tr>
<tr>
<td>ShowHelp</td>
<td>1036</td>
</tr>
<tr>
<td>ShowPopupHelp</td>
<td>1038</td>
</tr>
<tr>
<td>Sign</td>
<td>1039</td>
</tr>
<tr>
<td>SignalError</td>
<td>1039</td>
</tr>
<tr>
<td>Sin</td>
<td>1041</td>
</tr>
<tr>
<td>Sleep</td>
<td>1041</td>
</tr>
<tr>
<td>Sort</td>
<td>1042</td>
</tr>
<tr>
<td>SortAll</td>
<td>1042</td>
</tr>
<tr>
<td>Space</td>
<td>1043</td>
</tr>
<tr>
<td>Sqrt</td>
<td>1045</td>
</tr>
<tr>
<td>Start</td>
<td>1046</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>StartHotLink</td>
<td>1053</td>
</tr>
<tr>
<td>StartServerDDE</td>
<td>1054</td>
</tr>
<tr>
<td>State</td>
<td>1055</td>
</tr>
<tr>
<td>StepIt</td>
<td>1057</td>
</tr>
<tr>
<td>Stop</td>
<td>1057</td>
</tr>
<tr>
<td>StopHotLink</td>
<td>1059</td>
</tr>
<tr>
<td>StopServerDDE</td>
<td>1060</td>
</tr>
<tr>
<td>String</td>
<td>1060</td>
</tr>
<tr>
<td>String_To_Object</td>
<td>1066</td>
</tr>
<tr>
<td>SuspendTransaction</td>
<td>1070</td>
</tr>
<tr>
<td>SyntaxFromSQL</td>
<td>1071</td>
</tr>
<tr>
<td>SystemRoutine</td>
<td>1074</td>
</tr>
<tr>
<td>TabPostEvent</td>
<td>1075</td>
</tr>
<tr>
<td>TabTriggerEvent</td>
<td>1076</td>
</tr>
<tr>
<td>Tan</td>
<td>1077</td>
</tr>
<tr>
<td>Text</td>
<td>1077</td>
</tr>
<tr>
<td>TextLine</td>
<td>1078</td>
</tr>
<tr>
<td>Time</td>
<td>1079</td>
</tr>
<tr>
<td>Timer</td>
<td>1082</td>
</tr>
<tr>
<td>ToAnsi</td>
<td>1083</td>
</tr>
<tr>
<td>Today</td>
<td>1084</td>
</tr>
<tr>
<td>Top</td>
<td>1085</td>
</tr>
<tr>
<td>TotalColumns</td>
<td>1086</td>
</tr>
<tr>
<td>TotalItems</td>
<td>1086</td>
</tr>
<tr>
<td>TotalSelected</td>
<td>1087</td>
</tr>
<tr>
<td>ToUnicode</td>
<td>1088</td>
</tr>
<tr>
<td>TraceBegin</td>
<td>1088</td>
</tr>
<tr>
<td>TraceClose</td>
<td>1090</td>
</tr>
<tr>
<td>TraceDisableActivity</td>
<td>1091</td>
</tr>
<tr>
<td>TraceEnableActivity</td>
<td>1093</td>
</tr>
<tr>
<td>TraceEnd</td>
<td>1095</td>
</tr>
<tr>
<td>TraceError</td>
<td>1096</td>
</tr>
<tr>
<td>TraceOpen</td>
<td>1097</td>
</tr>
<tr>
<td>TraceUser</td>
<td>1099</td>
</tr>
<tr>
<td>TriggerEvent</td>
<td>1100</td>
</tr>
<tr>
<td>TriggerPBEvent</td>
<td>1102</td>
</tr>
<tr>
<td>Trim</td>
<td>1104</td>
</tr>
<tr>
<td>TrimW</td>
<td>1104</td>
</tr>
<tr>
<td>Truncate</td>
<td>1105</td>
</tr>
<tr>
<td>TrustVerify</td>
<td>1106</td>
</tr>
<tr>
<td>TypeOf</td>
<td>1109</td>
</tr>
<tr>
<td>Uncheck</td>
<td>1110</td>
</tr>
<tr>
<td>Undo</td>
<td>1112</td>
</tr>
<tr>
<td>UnitsToPixels</td>
<td>1113</td>
</tr>
</tbody>
</table>
UpdateLinksDialog ................................................................. 1113
Upper ...................................................................................... 1115
UpperBound .............................................................................. 1115
Which ...................................................................................... 1117
WordCap ............................................................................... 1119
WorkSpaceHeight ................................................................. 1119
WorkSpaceWidth ................................................................. 1121
WorkSpaceX ........................................................................... 1122
WorkSpaceY ........................................................................... 1123
Write ..................................................................................... 1124
XMLParseFile ......................................................................... 1125
XMLParseString ..................................................................... 1128
Year ....................................................................................... 1131
Yield ....................................................................................... 1132

Index ................................................................................................. 1135
About This Book

Audience
This book is for programmers who will use PowerBuilder® to build client/server or multitier applications.

How to use this book
This book describes syntax and usage information for the PowerScript® language including variables, expressions, statements, events, and functions.

Related documents
For a complete list of PowerBuilder documentation, see the preface of PowerBuilder Getting Started.

Other sources of information
Use the Sybase Getting Started CD, the SyBooks CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started 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 Getting Started 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.

- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks 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 SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

Refer to the SyBooks Installation Guide on the Getting Started CD, or the README.txt file on the SyBooks CD for instructions on installing and starting SyBooks.

- The Sybase Product Manuals Web site is an online version of the SyBooks 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 Sybase Developer Network.

To access the Sybase Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.
### Conventions

The formatting conventions used in this manual are:

<table>
<thead>
<tr>
<th>Formatting example</th>
<th>Indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve and Update</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 f_name</td>
</tr>
<tr>
<td></td>
<td>• User-defined objects such as dw_emp or w_main</td>
</tr>
<tr>
<td>variable or file name</td>
<td>When used in descriptive text and syntax descriptions, oblique font indicates:</td>
</tr>
<tr>
<td></td>
<td>• Variables, such as myCounter</td>
</tr>
<tr>
<td></td>
<td>• Parts of input text that must be substituted, such as pbiname.pbd</td>
</tr>
<tr>
<td></td>
<td>• File and path names</td>
</tr>
<tr>
<td>File&gt;Save</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>dw_1.Update()</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>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>
<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 the marker as a comment</td>
<td>Cannot extend to multiple lines</td>
</tr>
<tr>
<td>Slash and asterisk</td>
<td>/<em>...</em>/</td>
<td>Designate the text between the markers as a comment Nest comments</td>
<td>• Can extend over multiple lines (multiline comments do not require a continuation character) • Can be nested</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 User’s 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

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

**Identifier:**

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 134. For information about valid identifiers, see “Identifier names” on page 5.

**Examples**

On a line by itself above the statement

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

At the start of the line before the statement

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

Special ASCII characters

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.
CHAPTER 1 Language Basics

Syntax

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></td>
<td>Tab</td>
<td>~t</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertical tab</td>
<td>~v</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Carriage return</td>
<td>~r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Form feed</td>
<td>~f</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Backspace</td>
<td>~b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Double quote</td>
<td>~&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Single quote</td>
<td>~'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tilde</td>
<td>~~</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Any ASCII character</th>
<th>Decimal</th>
<th>~###</th>
<th>### = a 3-digit number from 000 to 255</th>
</tr>
</thead>
<tbody>
<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>Event</th>
<th>Next</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>event</td>
<td>next</td>
<td>shared</td>
</tr>
<tr>
<td>and</td>
<td>execute</td>
<td>not</td>
<td>static</td>
</tr>
<tr>
<td>autoinstantiate</td>
<td>exit</td>
<td>of</td>
<td>step</td>
</tr>
<tr>
<td>call</td>
<td>external</td>
<td>on</td>
<td>subroutine</td>
</tr>
<tr>
<td>case</td>
<td>false</td>
<td>open*</td>
<td>super</td>
</tr>
<tr>
<td>catch</td>
<td>fetch</td>
<td>or</td>
<td>system</td>
</tr>
<tr>
<td>choose</td>
<td>finally</td>
<td>parent</td>
<td>systemread</td>
</tr>
<tr>
<td>close*</td>
<td>first</td>
<td>post*</td>
<td>systemwrite</td>
</tr>
<tr>
<td>commit</td>
<td>for</td>
<td>prepare</td>
<td>then</td>
</tr>
<tr>
<td>connect</td>
<td>forward</td>
<td>prior</td>
<td>this</td>
</tr>
<tr>
<td>constant</td>
<td>from</td>
<td>private</td>
<td>throw</td>
</tr>
<tr>
<td>continue</td>
<td>function</td>
<td>privateread</td>
<td>throws</td>
</tr>
<tr>
<td>create*</td>
<td>global</td>
<td>privatewrite</td>
<td>to</td>
</tr>
<tr>
<td>cursor</td>
<td>goto</td>
<td>procedure</td>
<td>trigger</td>
</tr>
<tr>
<td>declare</td>
<td>halt</td>
<td>protected</td>
<td>true</td>
</tr>
<tr>
<td>delete</td>
<td>if</td>
<td>protectedread</td>
<td>try</td>
</tr>
<tr>
<td>describe*</td>
<td>immediate</td>
<td>protectedwrite</td>
<td>type</td>
</tr>
<tr>
<td>descriptor</td>
<td>indirect</td>
<td>prototypes</td>
<td>until</td>
</tr>
<tr>
<td>destroy</td>
<td>insert</td>
<td>public</td>
<td>update*</td>
</tr>
<tr>
<td>disconnect</td>
<td>into</td>
<td>readonly</td>
<td>updateblob</td>
</tr>
<tr>
<td>do</td>
<td>intrinsic</td>
<td>ref</td>
<td>using</td>
</tr>
<tr>
<td>dynamic</td>
<td>is</td>
<td>return</td>
<td>variables</td>
</tr>
<tr>
<td>else</td>
<td>last</td>
<td>rollback</td>
<td>while</td>
</tr>
<tr>
<td>elseif</td>
<td>library</td>
<td>rpcfunc</td>
<td>with</td>
</tr>
<tr>
<td>end</td>
<td>loop</td>
<td>select</td>
<td>within</td>
</tr>
<tr>
<td>enumerated</td>
<td>native</td>
<td>selectblob</td>
<td>_debug</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.
Pronouns

Description

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 1-4: PowerScript pronouns

<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 User’s 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 121.

Examples

Example 1  This example calls the ancestor function \text{wf_myfunc} (presumably the descendant also has a function called \text{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 User’s Guide.)

**Syntax**

```
Start of statement &
more statement &
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 17.

**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:

```powerbuilder
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:

```powerbuilder
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:

```
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”:

```
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:

Order - Balance // Subtracts Balance from Order
Order-Balance    // A variable named Order-Balance

Examples

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

A + B /* Adjustment factor */ + C

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

"The value of A + B is:"
White space
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>19</td>
</tr>
<tr>
<td>The Any datatype</td>
<td>24</td>
</tr>
<tr>
<td>System object datatypes</td>
<td>27</td>
</tr>
<tr>
<td>Enumerated datatypes</td>
<td>28</td>
</tr>
<tr>
<td>PowerBuilder datatypes in EAServer</td>
<td>29</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. The leading plus sign is not required (18 and +18 are the same). For example:

```
1  123   1200  +55   +1200
```

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 76.

**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 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:

\[
12.34 \quad 0.005 \quad 14.0 \quad -6500 \quad +3.5555
\]

Double

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

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:

\[
1 \quad 123 \quad 1200 \quad +55 \quad -32
\]

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 \(3.402822E+38\).

**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:

\[
2E4 \quad 2.5E78 \quad +6.02E3 \quad -4.1E-2 \\
-7.45E16 \quad 7.7E+8 \quad 3.2E-45
\]

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 76.

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:

```pascal
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:

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

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

```pascal
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~tIf(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

UnsignedLong or ULong

16-bit unsigned integers, from 0 to 65535.

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:

```PowerScript
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:

```PowerScript
any la_3
la_3 = ia_1 - ia_2
```

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

```PowerScript
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:

```plaintext
ls_string = ia_any
```

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

```plaintext
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*:

```plaintext
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:

```plaintext
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:

```plaintext
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
```
Enumerated datatypes

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:

```powerbuilder
// 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 78.

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 45.

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!:

```powerbuilder
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

All EAServer component interfaces are 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.

<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>31</td>
</tr>
<tr>
<td>Declaring constants</td>
<td>45</td>
</tr>
<tr>
<td>Declaring arrays</td>
<td>46</td>
</tr>
<tr>
<td>Declaring external functions</td>
<td>55</td>
</tr>
<tr>
<td>Declaring DBMS stored procedures as remote procedure calls</td>
<td>63</td>
</tr>
</tbody>
</table>

Declaring variables

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>
<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

General information
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:

```
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:

```
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:

```
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:

```
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`:

```
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”:

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

Naming errors If a local or global variable 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:

```powerbuilder
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:

```powerbuilder
{ access } datatype { ( size ) } { ( precision ) } variablename { = value } 
{ , variablename2 { = value2 } }
```

<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 40.</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 blobs and decimals, you can specify the size or precision of the data by including an optional value in brackets.</td>
</tr>
<tr>
<td>{ size }</td>
<td>(For blobs only) A number, enclosed in braces, specifying the size in bytes of the blob. If { size } 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>
<tr>
<td>(optional)</td>
<td></td>
</tr>
</tbody>
</table>
Declaring variables

Parameter | Description
---|---
/precision/ (optional) | (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.

variablename | 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.

value (optional) | 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 38.

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 User’s 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

Default 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>
<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 a 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 19.

Do not use a function’s return value

You should not initialize a variable by assigning it the return value of a function as in the following line of code, because it might not compile correctly:

```plaintext
integer i = f_return_one()
```

This example declares li_count as an integer whose value is 5:

```plaintext
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:

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

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

```powerscript
string ls_method="UPS"
```

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

```powerscript
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:

```powerscript
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:

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

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

```powerscript
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:

```powerscript
date d_date = Today( )
```

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

```powerscript
date d_date
d_date = Today( )
```
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 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

The general syntax for declaring PowerScript variables (see “Syntax of a variable declaration” on page 35) 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></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></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

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:

```plaintext
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.

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:

```plaintext
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:

protected privatewrite string is_label

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

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:

public privatered read privaterewrite 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`:

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):

```powerscript
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:

```powerscript
public privaterewrite 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):

```powerscript
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:

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:

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</td>
<td>(For instance variables only) Keywords specifying the access for the constant. For information, see “Access for instance variables” on page 40.</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 19.</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 38.</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:

```powerscript
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

\[
\{ \text{access} \} \text{datatype} \ \text{variablename} \ \{ d_1, \ldots, \ d_n \} \ \{ = \ \{ \text{valuelist} \} \}
\]

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 (optional)</td>
<td>(For instance variables only) Keywords specifying the access for the variable. For information, see “Access for instance variables” on page 40.</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 datatype (see “Syntax of a variable declaration” on page 35): decimal {2} \ variablename \ [ ] For blobs, fixed-length blobs within an array are not supported. If you specify a size after datatype, 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:

```plaintext
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.
```

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):

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

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

blob ib_image[10]      // Array of 10
                      // variable-size blobs.
dec(2) lc_Cost[10]    // Array of 10 decimal
                      // numbers.
                      // Each value has 2 digits
                      // following the decimal
                      // point.
decimal lc_price[20]  // Array of 20 decimal
                      // numbers.
                      // 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:

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

More declarations of multidimensional arrays:

```powerscript
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:

```powerscript
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:

```powerscript
integer li_TaxCode[3]
```


For information about default values for basic datatypes, see “Initial values for variables” on page 38.

**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:

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

Examples

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

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

**Example 2**  This statement initializes a two-dimensional array:

```powerbuilder
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

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

```powerbuilder
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,
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 arraylists 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
ingeninteger 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
ingeninteger 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
ingeninteger 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
geninteger 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
geninteger c[ ], a[2,2], b[3,3] = {1,2,3,4,5,6,7,8,9}
```

Array b is laid out like this:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

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

```plaintext
ageninteger a[2,10], b[2,10]
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
```

```plaintext

```

1 for b[1,1] 4 for b[1,2] 7 for b[1,3]
2 for b[2,1] 5 for b[2,2] 8 for b[2,3]
3 for b[3,1] 6 for b[3,2] 9 for b[3,3]
```
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 52.

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.
The second column on the second page has zeros:

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

```powerscript
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:

```powerscript
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

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:

```powerbuilder
{ 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:

```powerbuilder
{ 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</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.

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of a function or subroutine that resides in a DLL. Function names cannot contain special characters, such as the @ character, because they cause a compiler error. Use the ALIAS FOR clause described later in this table if the function name in the DLL contains special characters.</td>
</tr>
<tr>
<td>REF</td>
<td>A keyword that specifies that you are passing by reference the argument that follows REF. The function can store a value in arg that will be accessible to the rest of the PowerBuilder script.</td>
</tr>
<tr>
<td>datatype arg</td>
<td>The datatype and name of the arguments for the function or subroutine. The list must match the definition of the function in the DLL. Each datatype arg pair can be preceded by REF. For more information on passing arguments, see Application Techniques.</td>
</tr>
<tr>
<td>LIBRARY &quot;libname&quot;</td>
<td>A keyword followed by a string containing the name of the dynamic library in which the function or subroutine is stored. libname is a dynamic link library, which is a file that usually has the extension DLL on Windows. For components deployed to EAServer on UNIX, the file has an extension of .so, .sl, or .a, depending on the operating system.</td>
</tr>
<tr>
<td>ALIAS FOR &quot;extname&quot; (optional)</td>
<td>Keywords followed by a string giving the name of the function as defined in the dynamic library. If the name in the dynamic library is not the name you want to use in your script, or if the name in the database is not a legal PowerScript name, you must specify ALIAS FOR &quot;extname&quot; to establish the association between the PowerScript name and the external name.</td>
</tr>
<tr>
<td>.ansi</td>
<td>Required if the function passes a string as an argument or returns a string that uses ANSI encoding. Even if you use the default name for an ANSI function, you must always use the ALIAS keyword if you want to specify that the string uses ANSI encoding, because you must qualify the ALIAS with the .ansi keyword</td>
</tr>
</tbody>
</table>
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:

//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:

//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

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:

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:

```
RETURN GetSysColor (ai_index)
```

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

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

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

```
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><code>*</code> (any pointer)</td>
<td>32-bit pointer</td>
<td>Long</td>
</tr>
<tr>
<td><code>char *</code></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.
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.

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.

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 109.

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.

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):

```powerbuilder
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.

**Applies to**

Transaction object

**Syntax**

```powerbuilder
FUNCTION  rtndatatype functionname ( { REF } datatype1 arg1,..., { REF } datatypen argn ) RPCFUNC { ALIAS FOR “spname” }

SUBROUTINE functionname ( { REF } datatype1 arg1 , ...., { REF } datatypen argn ) RPCFUNC { ALIAS FOR “spname” }
```

<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"
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REF</td>
<td>Specifies that you are passing by reference the argument that follows REF.</td>
</tr>
<tr>
<td></td>
<td>The stored procedure can store a value in arg that will be accessible to</td>
</tr>
<tr>
<td></td>
<td>the rest of the PowerBuilder script.</td>
</tr>
<tr>
<td></td>
<td>When you pass a string by reference, all memory management is done in</td>
</tr>
<tr>
<td></td>
<td>PowerBuilder. The string variable must be long enough to hold the returned</td>
</tr>
<tr>
<td></td>
<td>value. To ensure that this is true, first declare the string variable, and</td>
</tr>
<tr>
<td></td>
<td>then use the Space function to fill the variable with blanks equal to the</td>
</tr>
<tr>
<td></td>
<td>maximum number of characters that you expect the function to return.</td>
</tr>
<tr>
<td>datatype arg</td>
<td>The datatype and name of the arguments for the stored procedure. The list</td>
</tr>
<tr>
<td></td>
<td>must match the definition of the stored procedure in the database. Each</td>
</tr>
<tr>
<td></td>
<td>datatype arg pair can be preceded by REF.</td>
</tr>
<tr>
<td>RPCFUNC</td>
<td>A keyword indicating that this declaration is for a stored procedure in a</td>
</tr>
<tr>
<td></td>
<td>DBMS, not an external function in a DLL. For information on declaring</td>
</tr>
<tr>
<td></td>
<td>external functions, see “Declaring external functions” on page 55.</td>
</tr>
<tr>
<td>ALIAS FOR &quot;spname&quot; (optional)</td>
<td>Keywords followed by a string naming the procedure in the database. If the</td>
</tr>
<tr>
<td></td>
<td>name in the database is not the name you want to use in your script or if</td>
</tr>
<tr>
<td></td>
<td>the name in the database is not a legal PowerScript name, you must specify</td>
</tr>
<tr>
<td></td>
<td>ALIAS FOR &quot;spname&quot; to establish the association between the PowerScript name</td>
</tr>
<tr>
<td></td>
<td>and the database name.</td>
</tr>
</tbody>
</table>
This code calls the function in a script:

```powerscript
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:

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

This code calls the `SPM8` stored procedure:

```powerscript
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
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>67</td>
</tr>
<tr>
<td>Operator precedence in PowerBuilder expressions</td>
<td>72</td>
</tr>
<tr>
<td>Datatype of PowerBuilder expressions</td>
<td>73</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

Description

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>Exponentation</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 118.

**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 73.

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:

\[ A\text{-}B \]

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

```powerbuilder
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
```

\[ d = 3  \times (20.0/3) \]  // d contains 20.0000
\[ e = \text{Truncate}(20.0/3, 4) \]  // e contains  6.6666
\[ f = \text{Truncate}(20.0/3, 5) \]  // 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:

```powerbuilder
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:

```powerbuilder
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 4-2: PowerBuilder 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=0.05</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>if Price&gt;100 then Rate=0.05</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>if Price&lt;100 then Rate=0.05</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
<td>if Price&lt;&gt;100 then Rate=0.05</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td>if Price&gt;=100 then Rate=0.05</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>if Price&lt;=100 then Rate=0.05</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical negation</td>
<td>if NOT Price=100 then Rate=0.05</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
<td>if Tax&gt;3 AND Ship &lt;5 then Rate=0.05</td>
</tr>
<tr>
<td>OR</td>
<td>Logical or</td>
<td>if Tax&gt;3 OR Ship&lt;5 then Rate=0.05</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 1115 and Lower on page 697.

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 891, LeftTrim on page 667, and Trim on page 1104.

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:

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

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
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 4-4: Order of precedence of operators*

<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 $(x + (y * (a + b)))$, $a + b$ 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

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:

```powerscript
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:

```powerscript
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:

- Convert integer \(a\) to long
- Convert integer \(b\) to long
- Add the longs \(a\) and \(b\)
- Convert the result from long to integer and assign the result to \(c\)
- Convert integer \(c\) to long and assign the result to \(e\)

The assignment to \(c\) causes the long result of the addition to be truncated, causing overflow and wrapping. Assigning \(c\) to \(e\) cannot restore the lost information.
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 \texttt{x} being assigned to the char variable \texttt{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 \texttt{c} to a string before comparison with the string \texttt{“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.
CHAPTER 5  Structures and Objects

About this chapter

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>77</td>
</tr>
<tr>
<td>About objects</td>
<td>78</td>
</tr>
<tr>
<td>Assignment for objects and structures</td>
<td>84</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 User’s 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

Reffering 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:

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:

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:

This.str_cust1.name

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

w_customer.str_cust1.name

About objects

What an object is

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.
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.

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.
About objects

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 User’s 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}
\texttt{w\_main \texttt{w\_1, w\_2}}
\texttt{Open(w\_1)}
\texttt{Open(w\_2)}
\end{verbatim}

You can also open windows by specifying the class in the Open function:

\begin{verbatim}
\texttt{window \texttt{w\_1, w\_2}}
\texttt{Open(w\_1, \"w\_main\")}
\texttt{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 CREATE statement:

\begin{verbatim}
\texttt{uo\_emp\_data uo\_1, uo\_2}
\texttt{uo\_1 = CREATE uo\_emp\_data}
\texttt{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 83.
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.
### Exceptions to garbage collection

There are a few objects that are prevented from being collected:

- **Visual objects**  
  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.

- **Timing objects**  
  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.

- **Shared objects**  
  Registered shared objects are not collected because the `SharedObjectRegister` function adds an internal reference. `SharedObjectUnregister` removes the internal reference.

### 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.

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 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).

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.
Assignment for objects and structures

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.

Assignment for structures

Declaring a structure variable creates an instance of that structure:

```pascal
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:

```pascal
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:

```pascal
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 104.

Assignment for objects

Declaring an object variable declares an object reference:

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

Using the CREATE statement creates an instance of the object:

```power_script
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:

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

Assignments between ancestor and descendant 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:

```power_script
uo_emp_data uo_emp1, uo_emp2
```

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

```power_script
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`:

```power_script
uo_emp2 = uo_emp1
```

For information about passing objects as function arguments, see “Passing arguments to functions and events” on page 104.
Assignment for objects and structures

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:

```
uo_emp_data uoempl, uoemp2 // Two object instances
```

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

```
uoempl = uoemp2
```

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 descendent 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:

```
uo_emp_data uoempl // Ancestor
uo_emp_active uoempa, uoempb // Descendants
uo_emp_inactive uoempi // 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:

uo_empb = uo_empa // Allowed, same type
uo_empa = uoemploi // Not allowed, different types

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

uo_emp1 = uo_empa

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

uo_empa = 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 104.
Assignment for objects and structures
CHAPTER 6

Calling Functions and Events

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>89</td>
</tr>
<tr>
<td>Finding and executing functions and events</td>
<td>92</td>
</tr>
<tr>
<td>Triggering versus posting functions and events</td>
<td>94</td>
</tr>
<tr>
<td>Static versus dynamic calls</td>
<td>96</td>
</tr>
<tr>
<td>Overloading, overriding, and extending functions and events</td>
<td>102</td>
</tr>
<tr>
<td>Passing arguments to functions and events</td>
<td>104</td>
</tr>
<tr>
<td>Using return values</td>
<td>107</td>
</tr>
<tr>
<td>Syntax for calling PowerBuilder functions and events</td>
<td>109</td>
</tr>
<tr>
<td>Calling functions and events in an object’s ancestor</td>
<td>112</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 6-1: Types of functions and events**

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Events</strong></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><strong>User event</strong></td>
<td></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 User's Guide.</td>
</tr>
<tr>
<td><strong>System or built-in event</strong></td>
<td></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><strong>Functions</strong></td>
<td>Function</td>
<td>A program or routine that performs specific processing.</td>
</tr>
<tr>
<td><strong>System function</strong></td>
<td></td>
<td>A built-in PowerScript function that is not associated with an object.</td>
</tr>
<tr>
<td><strong>Object function</strong></td>
<td></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><strong>User-defined function</strong></td>
<td></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><strong>Global function</strong></td>
<td></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><strong>Local external function</strong></td>
<td></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><strong>Global external function</strong></td>
<td></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><strong>Remote procedure call (RPC)</strong></td>
<td></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 102.

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:

```plaintext
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 descendent function.

For the syntax for calling ancestor functions, see “Calling functions and events in an object’s ancestor” on page 112.

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:
Triggering versus posting functions and events

**Triggering**

In PowerBuilder, when you trigger a function or event, it is called immediately. Its return value is available for use in the script.

**Posting**

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).

**PowerBuilder messages processed first**

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`.

**Restrictions for POST**

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 \texttt{w_activity_manager} window calls the functions \texttt{uf\_setup} and \texttt{uf\_set\_tabpgsystem}. (The functions belong to the user object \texttt{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 \texttt{tv\_roadmap} TreeView control in the \texttt{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 descendent is executed.
CHAPTER 6 Calling Functions and Events

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 \( \text{Set}(\text{integer}) \).
- Descendent window \( w_a\_desc \) with two functions: \( \text{Set}(\text{integer}) \) overrides the ancestor function, and \( \text{Set}(\text{string}) \) is an overload of the function.

Situation 1 Suppose you open the window \( \text{mywindow} \) of the ancestor window class \( w_a \):

\[
\begin{align*}
\text{w}_a & \ \text{mywindow} \\
\text{Open}(\text{mywindow})
\end{align*}
\]

This is what happens when you call the \text{Set} function statically or dynamically:

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

Situation 2 Now suppose you open \( \text{mywindow} \) as the descendant window class \( w_a\_desc \):

\[
\begin{align*}
\text{w}_a & \ \text{mywindow} \\
\text{Open}(\text{mywindow, } \text{"w_a\_desc"})
\end{align*}
\]
**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><code>mywindow.Set(1)</code></td>
<td>Compiles correctly because function is found in the ancestor <code>w_a</code>. At runtime, <code>Set(integer)</code> in the descendant is executed.</td>
</tr>
<tr>
<td><code>mywindow.Set(&quot;hello&quot;)</code></td>
<td>Fails to compile; no function prototype in the ancestor matches the call.</td>
</tr>
<tr>
<td><code>mywindow.DYNAMIC Set(&quot;hello&quot;)</code></td>
<td>Compiles successfully because of the DYNAMIC keyword. At runtime, <code>Set(string)</code> 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:

```powershell
guo_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`:

```powershell
guo_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:
   ```powerscript
   object.DYNAMIC funcname( )
   ```

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

3. This statement looks for an Any return value:
   ```powerscript
   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:
   ```powerscript
   object.EVENT DYNAMIC eventname( )
   ```

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

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

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 event does not exist.</td>
<td>The code <em>is not</em> 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>The code <em>is looking</em> for a return value.</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 <em>Any</em> in <em>Any</em> 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 <em>has a defined return value.</em></td>
<td>A null of the defined datatype is returned.</td>
<td>If <em>eventname</em> is defined to return integer, <em>li_int</em> is set to null in statement 2.</td>
</tr>
<tr>
<td></td>
<td>The event <em>does not have a defined return value.</em></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 <em>Any</em> in <em>Any</em> 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>

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

```plaintext
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 103.

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.

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
dramatically than expected if the change causes a different function to be called.

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 73.

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 descendant.
Passing arguments to functions and events

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 6-5: Passing arguments to functions and events

<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.

<table>
<thead>
<tr>
<th>Objects passed by value</th>
<th>Objects passed by reference</th>
<th>Objects passed as read-only</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Passing structures**

Structures as arguments behave like simple variables, not like objects.

<table>
<thead>
<tr>
<th>Structures passed by value</th>
<th>Structures passed by reference</th>
<th>Structures passed as read-only</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you pass a structure by value, PowerBuilder passes a copy of the structure. You can modify the copy without affecting the original.</td>
<td>When you pass a structure by reference, PowerBuilder passes a reference to the structure. When you change 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.</td>
<td>When you pass a structure as read-only, PowerBuilder passes a copy of the structure. You cannot modify any members of the structure.</td>
</tr>
</tbody>
</table>
Passing arrays

Variable-size array as an argument

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)
```

Fixed-size array as an argument

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 46.
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
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
dw_1.SelectRow(dw_1.GetRow(), true)
```

To ignore a return value, call the function as a single statement:

```power
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 92.</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 96. For more information on dynamic calls, see “Dynamic calls” on page 97.</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 94.</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>
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 112.

**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 108.

**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 55.
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`:

```powerscript
charnum = Asc("x")
```

This statement calls the DataWindow function in a script that belongs to the DataWindow:

```powerscript
Update()
```

This statement calls the global user-defined function `gf_setup_appl`:

```powerscript
gf_setup_appl(24, "Window1")
```

This statement calls the system function `PrintRect`:

```powerscript
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:

```powerscript
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:

```powerscript
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:

```powerscript
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:

```powerscript
This.EVENT DoubleClicked(xpos, ypos, row, dwo)
```

This statement posts the same event:

```powerscript
This.EVENT POST DoubleClicked(xpos, ypos, row, dwo)
```
**Example 5**  The variable `iw_a` is an instance variable of an ancestor window type `w_ancestorsheet`:

```powerbuilder
w_ancestorsheet iw_a
```

A menu has a script that calls the `wf_export` function, but that function is not defined in the ancestor. The `DYNAMIC` keyword is required so that the script compiles:

```powerbuilder
iw_a.DYNAMIC wf_export()
```

At execution time, the window that is opened is a descendant with a definition of `wf_export`. That window is assigned to the variable `iw_a` and the call to `wf_export` succeeds.

---

**Calling functions and events in an object’s ancestor**

**Description**

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.

**Syntax**

```powerbuilder
{ objectname } ancestorclass ::{ type } { when } name { { argumentlist } }
```

The following table describes the arguments used to call functions and events in an object’s ancestor.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>objectname</code></td>
<td>(optional) The name of the object whose ancestor contains the function you want to execute.</td>
</tr>
<tr>
<td><code>ancestorclass</code></td>
<td>The name of the ancestor class whose function or event you want to execute. The pronoun <code>Super</code> provides the appropriate reference when <code>ancestorobject</code> is the immediate ancestor of the current object.</td>
</tr>
<tr>
<td><code>type</code></td>
<td>(optional) A keyword specifying whether you are calling a function or event. Values are:</td>
</tr>
<tr>
<td></td>
<td>• (Default) FUNCTION</td>
</tr>
<tr>
<td></td>
<td>• EVENT</td>
</tr>
</tbody>
</table>

---

PowerBuilder
CHAPTER 6  Calling Functions and Events

Usage

The AncestorReturnValue variable  When you extend an event script in a descendent 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 descendent 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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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:</td>
</tr>
<tr>
<td></td>
<td>• TRIGGER – (Default) Execute it immediately</td>
</tr>
<tr>
<td></td>
<td>• POST – Put it in the object’s queue and execute it in its turn, after other pending messages have been handled</td>
</tr>
<tr>
<td>name</td>
<td>The name of the object 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 definition.</td>
</tr>
</tbody>
</table>
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:

\[
\text{ancestor}\_\text{name}::\text{EVENT event\_name}( )
\]

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.

```powerbuilder
// execute code that does some preliminary processing
CALL SUPER::uo_myevent
IF AncestorReturnValue = 1 THEN
...
```

For information about CALL, see CALL on page 121.

### Examples

**Example 1** Suppose a window \texttt{w_ancestor} has an event \texttt{ue_process}. A descendant 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 descendant script. If the descendant overrides the ancestor, it calls the descendant script only:

```powerbuilder
EVENT ue\_process( )
```

This statement calls the ancestor event only (this script works if the calling script belongs to another object or the descendant window):

```powerbuilder
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.

```powerbuilder
Super::EVENT Clicked(0, x, y)
```

**Example 3** These statements call a function \texttt{wf\_myfunc} in the ancestor window (presumably, the descendant also has a function called \texttt{wf\_myfunc}):

```powerbuilder
Super::wf\_myfunc( )
Super::POST wf\_myfunc( )
```
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>118</td>
</tr>
<tr>
<td>CALL</td>
<td>121</td>
</tr>
<tr>
<td>CHOOSE CASE</td>
<td>122</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>124</td>
</tr>
<tr>
<td>CREATE</td>
<td>125</td>
</tr>
<tr>
<td>DESTROY</td>
<td>128</td>
</tr>
<tr>
<td>DO...LOOP</td>
<td>129</td>
</tr>
<tr>
<td>EXIT</td>
<td>131</td>
</tr>
<tr>
<td>FOR...NEXT</td>
<td>132</td>
</tr>
<tr>
<td>GOTO</td>
<td>134</td>
</tr>
<tr>
<td>HALT</td>
<td>135</td>
</tr>
<tr>
<td>IF...THEN</td>
<td>136</td>
</tr>
<tr>
<td>RETURN</td>
<td>138</td>
</tr>
<tr>
<td>THROW</td>
<td>139</td>
</tr>
<tr>
<td>THROWS</td>
<td>140</td>
</tr>
<tr>
<td>TRY...CATCH...FINALLY...END TRY</td>
<td>141</td>
</tr>
</tbody>
</table>
Assignment

Description
Assigns values to variables or object properties or object references to object variables.

Syntax
\[ \text{variablename} = \text{expression} \]

<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. \text{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 \text{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:

\[
\begin{align*}
\text{String1} &= \text{"Part is out of stock"} \\
\text{TaxRate} &= .05
\end{align*}
\]

\text{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:

\[
\text{A} = \text{B} = 0 \quad // \text{This will not assign 0 to A and B.}
\]

This statement first evaluates \text{B=0} to \text{true} or \text{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.

\text{Assigning array values} You can assign multiple array values with one statement, such as:

\[
\begin{align*}
\text{int Arr[]} \\
\text{Arr} &= \{1, 2, 3, 4\}
\end{align*}
\]

You can also copy array contents. For example, this statement copies the contents of \text{Arr2} into array \text{Arr1}:

\[
\text{Arr1} = \text{Arr2}
\]
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:

```powerpoint
    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:

```powerpoint
    window lw_current_window
    lw_current_window = Parent
```

Example 3 This statement makes a CheckBox invisible:

```powerpoint
    cbk_on.Visible = FALSE
```

Example 4 This statement is not an assignment—it tests the value of the string in the SingleLineEdit sle_emp:

```powerpoint
    IF sle_emp.Text = "N" THEN Open(win_1)
```
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:

```cpp
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:

```cpp
int i, j
i = 12
j = i ++   // INVALID
```

The following is valid, because ++ is used by itself in the assignment:

```cpp
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 179.

Syntax
CALL ancestorobject {`controlname`}::event

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 109.

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 descendent 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** | **Description**
--- | ---
`testexpression` | The expression on which you want to base the execution of the script.
`expressionlist` | One of the following expressions:
- A single value
- A list of values separated by commas (such as 2, 4, 6, 8)
- A **TO** clause (such as **1 TO 30**)
- **IS** followed by a relational operator and comparison value (such as **IS>5**)
- Any combination of the above with an implied **OR** between expressions (such as **1, 3, 5, 7, 9, 27 TO 33, IS >42**)
`statementblock` | The block of statements you want PowerBuilder to execute if the test expression matches the value in `expressionlist`.

**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*:

```
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:

```
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 48.9999"
    CASE is > 48.9999
        sle_message.Text = "Real Case > 48.9999"
    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 131.

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:

```plaintext
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:

```plaintext
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):

\[
\text{objectvariable} = \text{CREATE} \ \text{objecttype}
\]

Syntax 2 (allows the application to choose the object type dynamically):

\[
\text{objectvariable} = \text{CREATE USING} \ \text{objecttypestring}
\]

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 800.)

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:

```
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 82.

**Examples**

**Example 1**  These statements create a new transaction object and stores the object in the variable DBTrans:

```
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:

```powerScript
n_service invo_service
```

The Open event for the object creates the service object:

```powerScript
//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:

```powerScript
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:

```powerScript
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:

```powerScript
l_ds_delete = CREATE u_ds
l_ds_delete.DataObject = 'd_user_delete'
l_ds_delete.SetTransObject(SQLCA)
li_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:

```powerScript
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 82.

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>
**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 124.

Examples
Example 1  This EXIT statement causes the loop to terminate if an element in the Nbr array equals 0:

```power
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:

```power
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

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:

```powerbuilder
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.

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>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.</td>
</tr>
<tr>
<td>start</td>
<td>Starting value of varname.</td>
</tr>
<tr>
<td>end</td>
<td>Ending value of varname.</td>
</tr>
<tr>
<td>increment</td>
<td>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.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
</tr>
<tr>
<td>statementblock</td>
<td>The block of statements you want to repeat.</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>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.</td>
</tr>
<tr>
<td>start</td>
<td>Starting value of varname.</td>
</tr>
<tr>
<td>end</td>
<td>Ending value of varname.</td>
</tr>
<tr>
<td>increment</td>
<td>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.</td>
</tr>
<tr>
<td>statementblock</td>
<td>The block of statements you want to repeat.</td>
</tr>
</tbody>
</table>
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 $\geq 5$ and $\leq 25$:

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 $\geq 5$ and $\leq 25$:

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
    .
    .
    .
    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, COM+, 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`:

```
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`:

```
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):

```
IF condition THEN action1 {ELSE action2}
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>The condition you want to test.</td>
</tr>
<tr>
<td>action1</td>
<td>The action you want performed if the condition is <strong>true</strong>. The action must be a single statement on the same line as the rest of the <strong>IF</strong> statement.</td>
</tr>
<tr>
<td>action2 (optional)</td>
<td>The action you want performed if the condition is <strong>false</strong>. The action must be a single statement on the same line as the rest of the <strong>IF</strong> statement.</td>
</tr>
</tbody>
</table>

Syntax 2 (the multiline format):

```
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>condition1</td>
<td>The first condition you want to test.</td>
</tr>
<tr>
<td>action1</td>
<td>The action you want performed if <strong>condition1</strong> is <strong>true</strong>. 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>condition2 (optional)</td>
<td>The condition you want to test if <strong>condition1</strong> is <strong>false</strong>. You can have multiple ELSEIF...THEN statements in an IF...THEN control structure.</td>
</tr>
<tr>
<td>action2</td>
<td>The action you want performed if <strong>condition2</strong> is <strong>true</strong>. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines.</td>
</tr>
<tr>
<td>action3 (optional)</td>
<td>The action you want performed if none of the preceding conditions is <strong>true</strong>. 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 \texttt{w\_first} if \texttt{Num} is equal to 1; otherwise, \texttt{w\_rest} is opened:

\begin{verbatim}
IF Num = 1 THEN Open(w\_first) ELSE Open(w\_rest)
\end{verbatim}

Example 2  This single-line IF...THEN statement displays a message if the value in the SingleLineEdit \texttt{sle\_State} is “TX”. It uses the continuation character to continue the single-line statement across two physical lines in the script:

\begin{verbatim}
IF sle\_State.text="TX" THEN &
    MessageBox("Hello","Tex")
\end{verbatim}

Example 3  This multiline IF...THEN compares the horizontal positions of windows \texttt{w\_first} and \texttt{w\_second}. If \texttt{w\_first} is to the right of \texttt{w\_second}, \texttt{w\_first} is moved to the left side of the screen:

\begin{verbatim}
IF w\_first.X > w\_second.X THEN
    w\_first.X = 0
END IF
\end{verbatim}

Example 4  This multiline IF...THEN causes the application to:

- Beep twice if \texttt{X} equals \texttt{Y}
- Display the Parts list box and highlight item 5 if \texttt{X} equals \texttt{Z}
- Display the Choose list box if \texttt{X} is blank
- Hide the Empty button and display the Full button if none of the above conditions is true

\begin{verbatim}
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
\end{verbatim}
RETURN

Description
Stops the execution of a script or function immediately.

Syntax
RETURN { expression }

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```
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:

```
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 exvalue`

Parameter | Description
---|---
`exvalue` | 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

\[ \text{methodname} \ (\ \{ \text{arguments} \} \ ) \ \text{THROWS} \ \text{ExceptionType} \ (\ , \ \text{ExceptionType}, \ ... \ ) \]

<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 User’s 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

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trystatements</td>
<td>Block of code that might potentially throw an exception.</td>
</tr>
<tr>
<td>ThrowableTypeN</td>
<td>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.</td>
</tr>
<tr>
<td>exIdentifierN</td>
<td>Local variable of type ThrowableTypeN.</td>
</tr>
<tr>
<td>catchstatementsN</td>
<td>Code to handle the exception being caught.</td>
</tr>
<tr>
<td>cleanupstatements</td>
<td>Cleanup code. The FINALLY block is optional if you include one or more CATCH block.</td>
</tr>
</tbody>
</table>
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>144</td>
</tr>
<tr>
<td>CLOSE rsor</td>
<td>147</td>
</tr>
<tr>
<td>CLOSE Procedure</td>
<td>148</td>
</tr>
<tr>
<td>COMMIT</td>
<td>149</td>
</tr>
<tr>
<td>CONNECT</td>
<td>150</td>
</tr>
<tr>
<td>DECLARE Cursor</td>
<td>151</td>
</tr>
<tr>
<td>DECLARE Procedure</td>
<td>151</td>
</tr>
<tr>
<td>DELETE</td>
<td>153</td>
</tr>
<tr>
<td>DELETE Where Current of Cursor</td>
<td>154</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>154</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>155</td>
</tr>
<tr>
<td>FETCH</td>
<td>156</td>
</tr>
<tr>
<td>INSERT</td>
<td>157</td>
</tr>
<tr>
<td>OPEN Cursor</td>
<td>158</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>158</td>
</tr>
<tr>
<td>SELECT</td>
<td>159</td>
</tr>
<tr>
<td>SELECTBLOB</td>
<td>160</td>
</tr>
<tr>
<td>UPDATE</td>
<td>162</td>
</tr>
<tr>
<td>UPDATEBLOB</td>
<td>163</td>
</tr>
<tr>
<td>UPDATE Where Current of Cursor</td>
<td>164</td>
</tr>
<tr>
<td>Using dynamic SQL</td>
<td>165</td>
</tr>
<tr>
<td>Dynamic SQL Format 1</td>
<td>168</td>
</tr>
<tr>
<td>Dynamic SQL Format 2</td>
<td>169</td>
</tr>
<tr>
<td>Dynamic SQL Format 3</td>
<td>170</td>
</tr>
<tr>
<td>Dynamic SQL Format 4</td>
<td>173</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:

```sql
INSERT INTO EMPLOYEE ( SALARY )
VALUES ( 18900 ) ;
```

The same statement using a PowerScript variable to reference the constant might look like this:

```powerscript
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:

```powerscript
:Name, :Address, :City
```

The same `HostVariableList` with indicator variables looks like this:

```powerscript
: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 \texttt{IndVar2} to see if \texttt{Address} contains a null value:

\begin{verbatim}
  if IndVar2 = -1 then...
\end{verbatim}

You can also use the PowerScript \texttt{IsNull} function to accomplish the same result without using indicator variables:

\begin{verbatim}
  if IsNull( Address ) then ... 
\end{verbatim}

This statement uses the indicator variable \texttt{IndVar3} to set \texttt{City} to null:

\begin{verbatim}
  IndVar3 = -1
\end{verbatim}

You can also use the PowerScript \texttt{SetNull} function to accomplish the same result without using indicator variables:

\begin{verbatim}
  SetNull( City )
\end{verbatim}

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 \texttt{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 \texttt{SQLErrText} or \texttt{SQLDBCode} to obtain the detail.</td>
</tr>
</tbody>
</table>

After certain statements, such as \texttt{DELETE}, \texttt{FETCH}, and \texttt{UPDATE}, you should also check the \texttt{SQLNRows} property of the transaction object to make sure the action affected at least one row.

\textbf{About \texttt{SQLErrText} and \texttt{SQLDBCode}} The string \texttt{SQLErrText} in the transaction object contains the database vendor-supplied error message. The long named \texttt{SQLDBCode} in the transaction object contains the database vendor-supplied status code:

\begin{verbatim}
  IF SQLCA.SQLCode = -1 THEN 
    MessageBox("SQL error", SQLCA.SQLErrText)
  END IF
\end{verbatim}
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 32.

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:

```plaintext
string res
select user_name() into:res from dummy;
select db_name() into:res from dummy;
select 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 add a new entry to the PB.INI file:

```plaintext
[pb]
dbsign=0
```
Caution
Set the value of dbsign to 0 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 rsor

Description
Closes the SQL cursor CursorName; ends processing of CursorName.

Syntax
CLOSE CursorName;

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};

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransactionObject</td>
<td>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).</td>
</tr>
</tbody>
</table>

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:

    COMMIT ;

Example 2  This statement commits all operations for the database specified in the transaction object named Emp_tran:

    COMMIT USING Emp_tran ;
CONNECT

 CONNECT

Description
Connects to a specified database.

Syntax
CONNECT [USING TransactionObject];

<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>

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 ;
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 32.

**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 32.

### 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
Deletes the rows in TableName specified by Criteria.

Syntax
DELETE FROM TableName WHERE Criteria {USING TransactionObject};

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TableName</td>
<td>The name of the table from which you want to delete rows.</td>
</tr>
<tr>
<td>Criteria</td>
<td>Criteria that specify which rows to delete.</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 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:

    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:

    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

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><code>Cursor or Procedure</code></td>
<td>The name of the cursor or procedure from which you want to fetch a row</td>
</tr>
<tr>
<td><code>HostVariableList</code></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 `SingleLineEdit`, 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:

```power_script
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:

```power_script
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).
CHAPTER 8  SQL Statements

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.
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**

```
SELECTBLOB RestOfSelectStatement (USING TransactionObject);
```

**Parameter** | **Description**
---|---
`RestOfSelectStatement` | The rest of the `SELECT` statement (the `INTO`, `FROM`, and `WHERE` 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 `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):

```plaintext
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_id_pic)
```

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>TableName</td>
<td>The name of the table in which you want to update rows.</td>
</tr>
<tr>
<td>RestOfUpdateStatement</td>
<td>The rest of the UPDATE statement (the SET 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**

**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} ;

Parameter | Description
--- | ---
TableName | The name of the table you want to update.
BlobColumn | The name of the column you want to update in TableName. The datatype of this column must be blob.
BlobVariable | A PowerScript variable of the datatype blob.
RestOfUpdateStatement | The rest of the UPDATE statement (the WHERE clause).
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

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:

```PowerScript
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;
```

<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 the row</td>
</tr>
<tr>
<td><code>SetStatement</code></td>
<td>The word <code>SET</code> followed by a comma-separated list of the form <code>ColumnName = value</code></td>
</tr>
<tr>
<td><code>CursorName</code></td>
<td>The name of the cursor in which the table is referenced</td>
</tr>
</tbody>
</table>

**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.
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 execution time. This is called dynamic SQL. The parameters used in dynamic SQL statements can change each time the program is executed.

Using Adaptive Server® Anywhere


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.
Using dynamic SQL

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:

```sql
DynamicDescriptionArea dda_desc1
dsa_desc1 = CREATE DynamicDescriptionArea
```

For more information about SQLDA, see Dynamic SQL Format 4 on page 173.

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:

```sql
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:

```sql
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
Dynamic SQL Format 1

If you have multiple PREPARE statements, the order affects the contents of SQLSA.

These statements illustrate the correct ordering:

```
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.
```

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 Employee. 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.

```pascal
string Mysql
Mysql = "CREATE TABLE Employee "
+ "(emp_id integer not null,"
+ "dept_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:

```pascal
string Mysql
Mysql="INSERT INTO dept Values (1234, 'Purchasing')"
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} ;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DynamicStagingArea</td>
<td>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.</td>
</tr>
</tbody>
</table>
Dynamic SQL Format 3

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 two parameters in SQLSA and then execute it using the value of the PowerScript variables Dept_id_var and Dept_name_var (note that Dept_name_var is null):

```
INT Dept_id_var = 156
String Dept_name_var
SetNull(Dept_name_var)
PREPARE SQLSA
    FROM "INSERT INTO dept VALUES (?,?)" ;
EXECUTE SQLSA USING :Dept_id_var,:Dept_name_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 ;

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.

<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). 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. 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>The name of the transaction object that identifies the database. (optional)</td>
</tr>
<tr>
<td>ParameterList</td>
<td>A comma-separated list of PowerScript variables. Note that PowerScript variables are preceded by a colon (:). (optional)</td>
</tr>
<tr>
<td>HostVariableList</td>
<td>The list of PowerScript variables into which the data values will be retrieved.</td>
</tr>
</tbody>
</table>
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 32.

**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`:

```powershell
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`:

```powershell
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 emp_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`:

```powershell
// The syntax of emp_select is:
// "SELECT emp_id
// FROM employee WHERE emp_state=@stateparm".
DECLARE my_proc DYNAMIC PROCEDURE FOR SQLSA;
```

172
integer Emp_id_var
string Emp_state_var

PREPARE SQLSA FROM "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 ;

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 ;

<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). 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>
</tbody>
</table>
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 32.

**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>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLStatement</td>
<td>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.</td>
</tr>
<tr>
<td>TransactionObject (optional)</td>
<td>The name of the transaction object that identifies the database.</td>
</tr>
<tr>
<td>DynamicDescriptionArea</td>
<td>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.</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
- GetDynamicNumber
- GetDynamicString
- GetDynamicTime

For information about these functions, see GetDynamicDate on page 520, GetDynamicDateTime on page 522, GetDynamicNumber on page 524, GetDynamicString on page 525, and GetDynamicTime on page 526.

Parameter values  The following enumerated datatypes are the valid values for the input and output parameter types:

- TypeBoolean!
- TypeDate!
- TypeDateTime!
- TypeDecimal!
- TypeDouble!
- TypeInteger!
- TypeLong!
- 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.

**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:

```
string Stringvar, Sqlstatement
integer Intvar
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
// an value of the enumerated datatype ParmType
// (such as TypeInteger!, or TypeString!).
CHOOSE CASE SQLDA.OutParmType[1]
    CASE TypeString!
        Stringvar = GetDynamicString(SQLDA, 1)
    CASE TypeInteger!
        Intvar = GetDynamicNumber(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
Sqlstatement = "SELECT emp_id FROM employee "
    +"WHERE emp_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.

CLOSE my_cursor;
CHAPTER 9

PowerScript Events

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.

    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 121.

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:

    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 112.

### 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.

For more information

If you want to trigger events, including system events, see “Syntax for calling PowerBuilder functions and events” on page 109 for information on the calling syntax.

To learn more about user-defined events, see the PowerBuilder User’s 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
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>

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 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.
BeginDrag

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:

   // 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:

   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
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:

```power
code
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:

```power
code
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

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_lvnbeginelabeledit</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:

```powershell
ListItem lvi
This.GetItem(index lvi)
slc_info.text = "Editing " + string(lvi.label)
```

See also

EndLabelEdit
**Syntax 2**

**For TreeView controls**

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

---

**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>
**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

188  PowerBuilder
**Syntax 2**

For **TreeView controls**

**Description**

Occurs when the user presses the right mouse button in the TreeView control and begins dragging.

**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>

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnbeginsdrag</td>
<td>TreeView</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
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

Description
Occurs when the user chooses an item on a menu.

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
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:

```powershell
/* Create a new sheet */
w_genapp_frame.wf_newsheet()
```

See also
Selected
CHAPTER 9  PowerScript Events

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 index 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.

Examples  This code changes the label of the item the user clicks to uppercase:

```powerScript
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
Clicked

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_tnclicked</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:

```powerbuilder
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:

```power brink
TreeViewItem ltvi_current

This.GetItem(handle, ltvi_current)
ltvi_current.Label = Upper(ltvi_current.Label)
This.SetItem(handle, ltvi_current)
```

See also
DoubleClicked
RightClicked
RightDoubleClicked
SelectionChanged
SelectionChanging
**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    | UnsignedLong 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 254. |
| 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:

```
IF (xpos <= 600 AND ypos <= 600) THEN
    cb_clear.SetFocus()
END IF
```

See also

DoubleClick
MouseDown
MouseMove
MouseUp
RButtonDown

Syntax 6

For other controls

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,</td>
</tr>
<tr>
<td></td>
<td>PictureHyperLink, PictureButton, RadioButton,</td>
</tr>
<tr>
<td></td>
<td>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
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**

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**

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:

```
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:

```
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 update 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 an 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
```
sle_info.Text = String(column) &
    + " " + ls_label &
    + " " + ls_align &
    + " " + String(li_width)

See also
    Clicked

**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
**DataChange**

- 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:

```plaintext
dw_1.SetTransObject(SQLCA)
dw_1.Retrieve()
```

**See also**

Destructor
Open

---

**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_onmndatachange</td>
<td>OLE</td>
</tr>
</tbody>
</table>

**Arguments**

None

**Return codes**

Long. Return code: Ignored

**See also**

PropertyRequestEdit
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.

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:

```powerShell
date dt_selected
integer li_ret
string ls_date

li_ret = GetSelectedDate( dt_selected)
ls_date = string(dt_selected)
sle_2.text = ls_date
```

See also
`DateChanged`
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):

0   Continue processing

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):

0   Continue processing

Examples: This example uses the DeleteAllItems event to ensure that there is a default item in the ListView control:

```plaintext```
This.AddItem("Default item", 1)
```

See also: DeleteItem, InsertItem
### Deleteltem

The Deleteltem 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 Deleteltem event displays a message with the number of the deleted item:

```plaintext
MessageBox("Message", "Item " + String(index) & 
            + " deleted.")
```

**See also**

- DeleteAllItems
- InsertItem
### 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

---

### Destructor

**Syntax 2**

**For TreeView controls**

**Description**

Occurs when an item is deleted.

**Event ID**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvousdeleteitem</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:

```powershell
TreeViewItem 11_tvi

This.GetItem(handle, 11_tvi)
MessageBox("Message", String(11_tvi.Label) & " has been deleted.")
```
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**

Occurs when the user double-clicks on the control.

**Event ID**

- pbm_lbdbleclk: ListBox, PictureListBox
- pbm_lvndoubleclicked: ListView
- pbm_tcdoubleclicked: Tab

**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:

```power-script
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:

```powerbuilder
TreeViewItem ltvi_current
ltvi_current = tv_1.FindItem(CurrentTreeItem!, 0)
This.EditLabel(ltvi_current)
```

See also
- Clicked
- RightClicked
- RightDoubleClicked
- SelectionChanged
- SelectionChanging
CHAPTER 9  PowerScript Events

Syntax 3

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_lbuttondblclk</td>
<td>Window</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>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 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 254.</td>
</tr>
<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>
<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):

| 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_cbndblck</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_lbuttondblck</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.
CHAPTER 9  PowerScript Events

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>
DragDrop

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:

```powershell
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
DragWithin

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, PictureBox, RadioButton</td>
</tr>
<tr>
<td>pbm_cbndragdrop</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_dragdrop</td>
<td>DatePicker, MonthCalendar</td>
</tr>
<tr>
<td>pbm_enddragdrop</td>
<td>SingleLineEdit, EditMask, MultiLineEdit, StaticText, StaticHyperLink</td>
</tr>
<tr>
<td>pbm_omndragdrop</td>
<td>OLE</td>
</tr>
<tr>
<td>pbm_prndragdrop</td>
<td>HProgressBar, VProgressBar</td>
</tr>
<tr>
<td>pbm_rndragdrop</td>
<td>RichTextEdit</td>
</tr>
<tr>
<td>pbm_sbndragdrop</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>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:

```powerbasic
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:

```powerbasic
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:

```powerbasic
DataWindow ldw_Source
Long ll_RowToDelete
Integer li_Choice
```
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_cbdragenter</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_dragenter</td>
<td>DatePicker, MonthCalendar</td>
</tr>
<tr>
<td>pbm_dwdragenter</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_lvdragenter</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_onndragenter</td>
<td>OLE</td>
</tr>
<tr>
<td>pbm_prndragenter</td>
<td>HProgressBar, VProgressBar</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:

```
IF source.TypeOf() = Picture! THEN
  This.Border = TRUE
END IF
```

See also

DragDrop
DragLeave
DragWithin

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, Picture, PictureHyperLink, PictureButton, RadioButton</td>
</tr>
<tr>
<td>pbm_cbdragleave</td>
<td>DropDownListBox, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_dragleave</td>
<td>DatePicker, MonthCalendar</td>
</tr>
<tr>
<td>pbm_dndragleave</td>
<td>DataWindow</td>
</tr>
</tbody>
</table>
DragLeave

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_endragleave</td>
<td>SingleLineEdit, EditMask, MultiLineEdit, StaticText, 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_rendragleave</td>
<td>RichTextEdit</td>
</tr>
<tr>
<td>pbm_sbdragleave</td>
<td>HScrollBar, HTrackBar, VScrollBar, VTrackBar</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:

```powerbuilder
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:

```powerbuilder
IF source.TypeOf() = Picture! THEN
  This.Border = TRUE
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.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_lbindragwithin</td>
<td>ListBox, PictureListBox</td>
</tr>
<tr>
<td>pbm_lvndragwithin</td>
<td>ListView</td>
</tr>
<tr>
<td>pbm_tcndragwithin</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>

**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:

```power_script
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>source</td>
<td>DragObject by value (a reference to the control being dragged)</td>
</tr>
<tr>
<td>handle</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

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_bndragwithin</td>
<td>CheckBox, CommandButton, Graph, InkEdit, InkPicture, Picture, PictureHyperLink, PictureButton, RadioButton</td>
</tr>
<tr>
<td>pbm_cbndragwithin</td>
<td>DropDownList, 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>
</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

*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.
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**

Description

Occurs when the user finishes editing an item’s label.

Event ID

- `pbm_lvnendlabeledit`
  - ListView

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>index</code></td>
<td>Integer. The index of the ListView item for which you have edited the label.</td>
</tr>
<tr>
<td><code>newlabel</code></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:

```pascal
ListViewItem lvi
sle_info.text = "Finished editing " &
               + String(lvi.label) &
               "+". Item changed to "+ String(newlabel)
```

See also

- BeginLabelEdit
CHAPTER 9  PowerScript Events

Syntax 2  For TreeView controls

Description
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_tvnenndeletedit</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>Integer. The index of the TreeView item for which you have edited the label.</td>
</tr>
<tr>
<td>newtext</td>
<td>The string that represents the new label for the TreeView 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
TreeViewItem tvi

This.GetItem(handle, tvi)
sle_info.Text = "Finished editing " &
   + String(tvi.Label) &
   + ". Item changed to " &
   + String(newtext)
```

See also
BeginLabelEdit
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 RuntimeError objects, such as DWRuntimeError or 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>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>errornumber</td>
<td>Unsigned integer by value (PowerBuilder’s error number)</td>
</tr>
<tr>
<td>errortext</td>
<td>String, read-only (PowerBuilder’s error message)</td>
</tr>
<tr>
<td>errorwindowmenu</td>
<td>String, read-only (the name of the window or menu that is the parent of the object whose script caused the error)</td>
</tr>
<tr>
<td>errorobject</td>
<td>String, read-only (the name of the object whose script caused the error)</td>
</tr>
<tr>
<td>errorscript</td>
<td>String, read-only (the full text of the script in which the error occurred)</td>
</tr>
<tr>
<td>errorline</td>
<td>Unsigned integer by value (the line in the script where the error occurred)</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</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 any active event handlers, or if none, 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. This option is not valid for DataWindows.</td>
</tr>
<tr>
<td></td>
<td>• ExceptionSubstituteReturnValue! – use the value specified in the returnValue argument instead of the value returned by the OLE server or DataWindow, and cancel the error condition.</td>
</tr>
<tr>
<td>returnValue</td>
<td>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!.</td>
</tr>
</tbody>
</table>
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:

```powerbuilder
    MessageBox("Error Number " + string(errornumber) &
    + " Occurred", "Errortext: " + String(errortext))
    action = ExceptionIgnore!
```

See also
DBError in the DataWindow Reference or the online Help
ExternalException
SystemError
# 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.

---

## Event ID

<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>
**ExternalException**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| action   | ExceptionAction by reference. A value you specify to control the application’s course of action as a result of the error. Values are:  
  • ExceptionFail! – fail as if this script were not implemented. The error condition triggers the SystemError event.  
  • 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).  
  • ExceptionRetry! – execute the function or evaluate the expression again in case the OLE server was not ready.  
  • ExceptionSubstituteReturnValue! – use the value specified in the returnValue argument instead of the value returned by the OLE server or DataWindow and cancel the error condition. |
| returnValue | 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 action is ExceptionSubstituteReturnValue!. |

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.
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:

```power
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:

```power
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:

```
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
```
CHAPTER 9  PowerScript Events

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.

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.

<table>
<thead>
<tr>
<th>Gesture</th>
<th>Gesture name</th>
<th>Argument value</th>
<th>Keystroke</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Left</td>
<td>0</td>
<td>Backspace</td>
</tr>
<tr>
<td></td>
<td>Right</td>
<td>1</td>
<td>Space</td>
</tr>
<tr>
<td></td>
<td>UpRightLong</td>
<td>2</td>
<td>Tab</td>
</tr>
<tr>
<td></td>
<td>DownLeftLong</td>
<td>3</td>
<td>Enter</td>
</tr>
<tr>
<td></td>
<td>UpRightLong</td>
<td>4</td>
<td>Tab</td>
</tr>
<tr>
<td></td>
<td>DownLeft</td>
<td>5</td>
<td>Enter</td>
</tr>
</tbody>
</table>

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:

```powerbuilder
CHOOSE CASE gest
CASE 0
  MessageBox("Gesture recognized", &
    "You entered a space")
CASE 1
  MessageBox("Gesture recognized", &
    "You entered a backspace")
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>
</tbody>
</table>
CHAPTER 9  PowerScript Events

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:

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:

This. EVENT POST ue_select( )

This code is in the ue_select user event:

This.SelectText(1, Len(This.Text))

See also
Clicked  LoseFocus

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<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_rensenfocus</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_lcsenfocus</td>
<td>Tab</td>
</tr>
<tr>
<td>pbm_tvnenfocus</td>
<td>TreeView</td>
</tr>
</tbody>
</table>
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 drags and clicks the question-mark button over a TrackBar control:

```pascal
MessageBox("Context Help Message", "Move the TrackBar" &
           + " slider to r-n 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:

    Idle(300)

In the Idle event itself, this statement closes the application:

    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:

    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:

```powerscript
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>
**ItemChanged**

**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>

**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:

```powerbuilder
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>
CHAPTER 9  PowerScript Events

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The index of the item that is changing</td>
</tr>
<tr>
<td>focuschanged</td>
<td>Boolean (specifies if focus has changed 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 has changed 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):

- **0**  Continue processing

Examples

This example checks whether the event is occurring because focus has changed to the item:

```power
ListViewItem l_lvi

lv_list.GetItem(index, l_lvi)
IF focuschange and hasfocus THEN
    s1e1.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>
</tbody>
</table>
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_tvncollapsed</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:

```powershell
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
```
CHAPTER 9  PowerScript Events

**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_tvitemcollapsing</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:

```powerScript
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
```

See also ItemCollapsing
ItemExpanded

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:

```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
```
CHAPTER 9  PowerScript Events

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

**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_tvnitemexpanding</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</td>
</tr>
<tr>
<td></td>
<td>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:

```
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
```
**ItemPopulate**

**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**

**Description**

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_tvntempopulate</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:

```plaintext
TreeViewItem tvi

This.GetItem(handle, tvi)
sle_get.Text = "Populating TreeView item " &  
+ String(tvi.Label) + " with children"
```

See also ItemExpanding
Key

Description
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>

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>
<tr>
<td>keyflags</td>
<td>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</td>
</tr>
</tbody>
</table>

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)
    END IF
END IF
```

This line displays the value of keyflags when a key is pressed.

```powerbuilder
st_1.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.

<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 st_1:

```powerScript
IF This.Position > This.MaxPosition - 1 THEN
    This.Position = MaxPosition
ELSE
    This.Position = This.Position + 1
END IF

st_1.Text = "LineDown " + String(This.Position)
```

See also

LineLeft
LineRight
LineUp
PageDown

---

**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_sblineup</td>
<td>HScrollBar, HTrackBar</td>
</tr>
</tbody>
</table>

**Arguments**

None

**Return codes**

Long. Return code choices (specify in a RETURN statement):

- 0  Continue processing
LineRight

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:

```powershell
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>pbn_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:

\[
\begin{align*}
\text{IF } & \text{This.Position} > \text{This.MaxPosition} - 1 \text{ THEN} \\
& \text{This.Position} = \text{MaxPosition} \\
\text{ELSE} & \text{This.Position} = \text{This.Position} + 1 \\
\text{END IF} \\
\end{align*}
\]

\[
\text{st_1.Text} = "LineRight " + \text{String(This.Position)}
\]

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
**LoseFocus**

**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:

```
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_bnkkillfocus</td>
<td>CheckBox, CommandButton, Graph, OLE, Picture, PictureHyperLink, PictureButton, RadioButton, StaticText, StaticHyperLink</td>
</tr>
<tr>
<td>pbm_chnkkillfocus</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>HProgressBar, 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>
</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:

```
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()
```
Modified

Description
Occurs when the contents in the control have changed.

Event ID

<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

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.

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
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:

```power-script
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>
### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>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 254.</td>
</tr>
<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>
<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

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:

```powerlanguage
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:

```powerlanguage
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**

Occurs when the mouse has moved within the RichTextEdit control.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renmousemove</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>

**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**

Occurs when the pointer is moved within the window.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_mousedownmove</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  
  *Flags* is the sum of all the buttons and keys that are pressed. |
| *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

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):

```powerShell
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
```
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_mousemove 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
```
// 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

Occurs when the user releases the left mouse button in a RichTextEdit control.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_renlbuttonup</td>
<td>RichTextEdit</td>
</tr>
</tbody>
</table>
MouseUp

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:

```
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>
<tbody>
<tr>
<td>flags</td>
<td>UnsignedLong by value (the modifier keys and mouse buttons that are pressed). Values are:</td>
</tr>
<tr>
<td></td>
<td>- 1 – Left mouse button</td>
</tr>
<tr>
<td></td>
<td>- 2 – Right mouse button</td>
</tr>
<tr>
<td></td>
<td>- 4 – Shift key</td>
</tr>
<tr>
<td></td>
<td>- 8 – Ctrl key</td>
</tr>
<tr>
<td></td>
<td>- 16 – Middle mouse button</td>
</tr>
<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>
<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):

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.

```plaintext
sle_2.Text = "Position of Pointer is: " + &
String(xpos) + "," + 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_modkey.

```plaintext
CHOOSE CASE flags
    CASE 0
        sle_mkey.Text = "No modifier keys pressed"
    CASE 4
        sle_mkey.Text = "SHIFT key pressed"
    CASE 8
        sle_mkey.Text = "CONTROL key pressed"
    CASE 12
        sle_mkey.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_sbnthumbacktrack</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:

```plaintext
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>
CHAPTER 9  PowerScript Events

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 366.

Examples
This example populates the SQLCA global variable from the application’s initialization file, connects to the database, and opens the main window:

```sql
/* 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**

Occurrences 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.

---

**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 \( \text{ii\_pagesize} \) 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):

```powerShell
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 \( \text{ii\_pagesize} \) and scrolls forward through a DataWindow control 10 rows for each page:

```powerShell
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.

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 \textit{ii_pagesize}):\[
\text{IF This\textunderscore Position < \& This\textunderscore MinPosition + ii\textunderscore pagesize THEN}
  \text{This\textunderscore Position = MinPosition}
\text{ELSE}
  \text{This\textunderscore Position = This\textunderscore Position - ii\textunderscore pagesize}
\text{ENDIF}
\]

See also
LineLeft
PageDown
PageRight
PageUp
CHAPTER 9  PowerScript Events

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):

<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 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 $ii_pagesize$):

```powerbasic
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
ENDIF
```

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
CHAPTER 9 PowerScript Events

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:

```powerscript
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:

```powershell
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:

```powershell
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):

<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  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.

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

Occurrences 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.

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>

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.
### 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><code>propertyname</code></td>
<td>String by value (the name of the property whose value changed).</td>
</tr>
<tr>
<td></td>
<td>If <code>propertyname</code> is an empty string, a more general change occurred, such as changes to more than one property.</td>
</tr>
<tr>
<td><code>cancelchange</code></td>
<td>Boolean by reference; determines whether the change will be canceled. Values are:</td>
</tr>
<tr>
<td></td>
<td>• FALSE – (Default) the change is allowed.</td>
</tr>
<tr>
<td></td>
<td>• TRUE – the change is canceled.</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 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
The RButtonDown event has different arguments for different objects:

### Syntax 1

**For controls and windows, except RichTextEdit**

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>
<tbody>
<tr>
<td>flags</td>
<td>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 254.</td>
</tr>
<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>
<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):

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):

```powerScript
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:

```powerScript
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_inkerecognition</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 ie_id InkEdit control is updated, then writes the recognized text to the string variable ls_inktext:

```powershell
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:

```powerbuilder
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:

```powerbuilder
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):

<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
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:

```c
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:

```c
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_ddrequest</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>

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>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1

**For ListView and Tab controls**

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):

<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 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:

```powerscript
// 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
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:

```pascal
// 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>
CHAPTER 9  PowerScript Events

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>
</table>
| index    | Integer by value (the index of the item or tab the user
double-clicked) |

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:

```powerScript
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
DoubleClick
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>
**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 choices (specify in a RETURN statement):

- 0  Continue processing

**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:

```powerbuilder
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.
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:

```power_script
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`:

```power_script
mle_menuhelp.Text = menuhelpstring
Timer(4)
```

This code in the Timer event clears the Help text and stops the timer:

```power_script
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>DropDownList, 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

Occurs when an item is selected in the control.

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_chnselchange</td>
<td>DropDownList, DropDownPictureListBox</td>
</tr>
<tr>
<td>pbm_lbselchange</td>
<td>ListBox, PictureListBox</td>
</tr>
</tbody>
</table>
SelectionChanged

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(i))
  END IF
NEXT

gr_1.SetRedraw(TRUE)
```

See also

Clicked
CHAPTER 9  PowerScript Events

Syntax 2  
For Tab controls

Description  
Occurs when a tab is selected.

Event ID  
<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tcnselchanged</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 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  
<table>
<thead>
<tr>
<th>Event ID</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbm_tvnselchanged</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 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
SelectionChanging

// 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_tcnsselchanging</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
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:

```powerbuilder
  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_tvselchanging</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):

- 0  Allow the selection to change
- 1  Prevent the selection from changing

Usage

The SelectionChanging event occurs before the SelectionChanged event.

Examples

This example displays the status of changing TreeView items in a SingleLineEdit:

```powerbuilder
  TreeViewItem l_tvinew, l_viol

  // Get TreeViewItem that was the old selection
```
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>
</tbody>
</table>
| status   | Long by value (the status of the window). Values are:  
- 0 – The current window is the only one affected.  
- 1 – The window’s parent is also being minimized or a pop-up window is being hidden.  
- 3 – The window’s parent is also being displayed or maximized or a pop-up window is being shown. |

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

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)
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

Occurs for each comparison when the ListView is being sorted.

**Event ID**

- pbm_lvnsort

**Objects**

- ListView

**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  *index1*  is less than  *index2*
0   *index1*  is equal to  *index2*
1   *index1*  is greater than  *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.

**Examples**

This example sorts ListView items according to PictureIndex and Label sorting by PictureIndex first, and then by label:

```power_script
ListViewItem lvi, lvi2
```
Syntax 2

For TreeView controls

Occurs for each comparison when the TreeView is being sorted.

<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

- **Argument**
  - **handle1**: Long by value (the handle of one item being compared during a sorting operation)
  - **handle2**: Long by value (the handle of the second item being compared)

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:

```powerScript
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
```

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):

- `0` Continue processing

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):
- 0  Continue processing

**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</td>
<td>InkEdit, InkPicture</td>
</tr>
<tr>
<td>pbm_inkpstroke</td>
<td></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
SystemError

Description
Occurs when a serious execution time 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 message box with the PowerBuilder error number and error message text. For information about error messages, see the PowerBuilder User's Guide.

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>
<tr>
<td>keyflags</td>
<td>UnsignedLong by value (the modifier keys that were pressed with the key). The only modifier key is the Shift key.</td>
</tr>
</tbody>
</table>

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:

```pascal
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.

```pascal
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):

- 0  Continue processing

**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:

```powerlanguage
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:

```powerlanguage
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:

```powerlanguage
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`:

```powershell
long ll_rowcount

ds_datastore.dataobject = "d_customers"
ds_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:

```powershell
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`:

```powershell
st_time.Text = String(Now(), "hh:mm")
```

---

### 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>pbn_tbnmoved</td>
<td>Window</td>
</tr>
</tbody>
</table>

**Arguments**

None
CHAPTER 9  PowerScript Events

Return codes
Long. Return code choices (specify in a RETURN statement):

0    Continue processing

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.

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.
ValueChanged

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:

```powerbuilder
IF IsDate(userstr) THEN
    dtm = DateTime(Date(userstr))
ELSE
    MessageBoxIcon("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:

```powerbuilder
MessageBox("Start date", "You selected " + &
    String(dtm, "mmm dd, yyyy") + ".-r-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
CHAPTER 10  PowerScript Functions

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

Contents
The functions are listed alphabetically.

See also
For information about functions that apply to DataWindows or DataStores, see also the DataWindow Reference. Methods that apply to DataWindows, but not to other PowerBuilder controls, are listed only in the DataWindow Reference.
**Abs**

Calculates the absolute value of a number.

**Syntax**

```
Abs ( n )
```

**Argument | Description**
--- | ---
`n` | The number for which you want the absolute value

**Return value**

The datatype of `n`. Returns the absolute value of `n`. If `n` is `null`, `Abs` returns `null`.

**Examples**

All these statements set `num` to 4:

```powershell
integer i, num

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

This statement returns 4.2:

```
Abs(-4.2)
```

**See also**

`Abs` method for DataWindows in the DataWindow Reference or online Help

---

**ACos**

Calculates the arccosine of an angle.

**Syntax**

```
ACos ( n )
```

**Argument | Description**
--- | ---
`n` | 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.

**Return value**

Double. Returns the arccosine of `n`.

**Examples**

This statement returns 0:

```
ACos(1)
```

This statement returns 3.141593 (rounded to six places):

```
ACos(-1)
```
This statement returns 1.000000 (rounded to six places):

\[ \text{ACos}(.540302) \]

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:

```powerbasic
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 DWObjects (objects within a DataWindow object that is within a DataWindow control)

### Syntax

```powerbasic
objectref.Activate ( activationtype )
```

<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 you want to activate. The fully qualified name for a DWObject has this syntax: <code>dwcontrol.Object.dwobjectname</code></td>
</tr>
</tbody>
</table>
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:

```powerbuilder
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:

```powerbuilder
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

AddColumn adds a column with a specified label, alignment, and width.

Applies to

ListView controls

Syntax

listviewname.AddColumn ( 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 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>
<tr>
<td>alignment</td>
<td>A value of the enumerated datatype Alignment specifying the alignment of the column you are adding. Values are: Center!, Justify!, Left!, Right!</td>
</tr>
<tr>
<td>width</td>
<td>An integer whose value is the width of the column you are adding, in PowerBuilder units.</td>
</tr>
</tbody>
</table>

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
AddData

Add a value to the end of a series in 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 Description**

<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</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. (optional)</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:

```powerbuilder
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:

```powerbuilder
integer SeriesNbr

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

See also

DeleteData
FindSeries
GetData
InsertData
Syntax 2  
For scatter graphs
Adds a data point to a series in a scatter graph.

Syntax
\( \text{controlname}.\text{AddData} \left( \text{seriesnumber}, \text{xvalue}, \text{yvalue} \right) \)

<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:

```powerScript
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>
**AddItem**

**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**  
```
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.

**Usage**  
If any argument’s value is null, AddItem returns null.

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:

```
integer rownbr
string s

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

If lb_Actions contains Add and Run and the Sorted property is false, the statement above returns 3 (because Edit File becomes the third and last item).

If the Sorted property is true, the statement above returns 2 (because Edit File becomes the second item after the list is sorted alphabetically).

**See also**  
DeleteItem
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>An integer specifying the index of the picture you want to associate with the newly added item (optional)</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 PictureList Box `plb_birds`:

```powerscript
integer li_pic, li_position
string ls_name, ls_pic

li_pic = plb_birds.AddItem("Cardinal")
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 )

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

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

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

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

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

See also
DeleteItem
FindItem
InsertItem
Reset
TotalItems
Syntax 4  
For ListView controls

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

Applies to
ListView controls

Syntax
```
listviewname.AddItem ( item )
```

<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 1_lvi
1_lvi.PictureIndex = 2
1_lvi.Label = "Item " + string(count)
lv_1.AddItem(1_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`:

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

See also
DeleteLargePicture
## AddPicture

**Description**

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

**Applies to**

PictureListBox, DropDownPictureListBox, and TreeView controls

**Syntax**

```
controlname.AddPicture(picturename)
```

<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:

```powerscript
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>listviewname</td>
<td>The name of the ListView control to which you are adding a small image</td>
</tr>
<tr>
<td>picturename</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**

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

**Applies to**
ListView and TreeView controls

**Syntax**

```
controlname.AddStatePicture ( picturename )
```

**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 AddPicture. You can change the StatePictureMaskColor property between calls.

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

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

**See also**
DeleteStatePicture
CHAPTER 10  PowerScript Functions

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**

**Description**
Arranges the icons in rows.

**Applies to**
ListView controls

**Syntax**

```
lv_list.Arrange ()
```

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

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

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

**Examples**
This example arranges the icons in a ListView control:

```
lv_list.Arrange()
```
**ArrangeSheets**

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

**Applies to**
MDI frame windows

**Syntax**
```
mdiframe. ArrangeSheets ( arrangetype )
```

**Argument** | **Description**
--- | ---
`mdiframe` | The name of an MDI frame window.
`arrangetype` | A value of the ArrangeTypes enumerated datatype specifying how you want the open sheets arranged in the MDI frame window. Values are:
- Cascade! – Cascade the sheets that are not minimized so that each sheet’s title bar is visible and arrange icons of minimized sheets in a row at the bottom of the frame.
- Layer! – Layer the sheets that are not minimized so that each sheet completely covers the one below it and arrange icons of minimized sheets in a row at the bottom of the frame.
- Tile! – Tile the sheets that are not minimized so that they do not overlap and arrange icons of minimized sheets in a row at the bottom of the frame.
- TileHorizontal! – Tile the sheets that are not minimized so that each is beside the other without overlapping and arrange icons of minimized sheets in a row at the bottom of the frame.
- Icons! – Arrange the minimized sheets in a row at the bottom of the frame.

**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**
```
Asc (string)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string for which you want the code point value of the first character</td>
</tr>
</tbody>
</table>

**Return value**
Unsigned Integer. Returns the code point value of the first character in `string`. If `string` is null, `Asc` returns null.

**Usage**
You can use `Asc` to find out the case of a character by testing whether its code point value is within the appropriate range. A separate function, `AscA`, is provided to return the Unicode code point of a character.

**Examples**
This statement returns 65, the code point value for uppercase A:
```
Asc("A")
```

This example checks if the first character of string `ls_name` is uppercase:
```
String ls_name
IF Asc(ls_name) > 64 and Asc(ls_name) < 91 THEN ...
```

**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**
```
AscA (string)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string for which you want the ASCII value of the first character</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns the ASCII value of the first character in `string`. If `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:

\[\text{AscA}("A")\]

This example checks if the first character of string \(ls\_name\) is uppercase:

\[
\text{String } \text{ls\_name} \\
\text{IF } \text{AscA}(\text{ls\_name}) > 64 \text{ and } \text{AscA}(\text{ls\_name}) < 91 \text{ THEN ...}
\]

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

\[
\text{string str\_from\_int, hold\_str} \\
\text{integer arraylen} \\
\text{arraylen = UpperBound(iarr)} \\
\text{FOR i = 1 to arraylen} \\
\quad \text{// Convert first character of pair to a char} \\
\quad \text{hold\_str = CharA(iarr[i])} \\
\quad \text{// Add characters to string after converting} \\
\quad \text{// the integer’s high byte to char} \\
\quad \text{str\_from\_int += hold\_str + \& CharA((iarr[i] - \text{AscA(hold\_str)}) / 256)} \\
\text{NEXT}
\]

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

See also

Asc
CharA
Mid
\text{AscA} \text{ 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:

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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>The ratio of the lengths of two sides of a triangle for which you want a corresponding angle (in radians)</td>
</tr>
</tbody>
</table>

**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 )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>The number of times you want the computer to beep. If <code>n</code> is greater than 10, the computer beeps 10 times.</td>
</tr>
</tbody>
</table>

**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**

```plaintext
CORBACurrent.BeginTransaction()  
```

**Argument** | **Description**
--- | ---
CORBACurrent | Reference to the CORBACurrent service instance

**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:

```plaintext
// 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
   11_rc = myconnect.CreateInstance(lcst_mybookstore)
   // begin processing
...

See also
CommitTransaction
GetContextService
GetStatus
GetTransactionName
Init
ResumeTransaction
RollbackOnly
RollbackTransaction
SetTimeout
SuspendTransaction

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>
<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
Blob. Returns the converted string in a blob with the requested encoding, if specified. If text is null, Blob returns null.

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.
This example saves a text string as a Unicode blob:

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

This example saves a text string as a blob with UTF-8 encoding:

```powerbuilder
Blob Blb
Blb = Blob("Any Text", EncodingUTF8!)
```

### BlobEdit

**Description**
Inserts data of any PowerBuilder datatype into a blob variable.

**Syntax**

```powerbuilder
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 blob variable 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:

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

... // Read BMP file containing employee picture
... // into temp using FileOpen and FileRead.
pic_date = Today()

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

This example copies a string into the blob blb_data starting at position 1 and specifies that the blob should use ANSI encoding:

```powerscript
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>data</td>
<td>Data of the blob datatype</td>
</tr>
<tr>
<td>n</td>
<td>The number (1 to 4,294,967,295) of the first byte you want returned</td>
</tr>
<tr>
<td>length (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 datablob starting at position 5 in the blob data_1; the second call stores the bytes of datablob from position 5 to the end in data_2:

```
blob data_1, data_2, datablob
... // Read a blob datatype into datablob.

data_1 = BlobMid(datablob, 5, 10)
data_2 = BlobMid(datablob, 5)
```

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

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

... // Read BMP file containing employee picture
... // into temp using FileOpen and FileRead.
```
pic_date = Today()
nbr = BlobEdit(emp_photo, 1, temp)
BlobEdit(emp_photo, nbr, pic_date)
st_1.Text = String(Date(BlobMid(emp_photo, nbr)))

See also
Blob
BlobEdit

BuildModel

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

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

Applies to
Profiling and TraceTree objects

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>Instance name of the Profiling or TraceTree object</td>
</tr>
<tr>
<td>progressobject</td>
<td>A PowerObject that represents the number of activities that have been processed</td>
</tr>
<tr>
<td>eventname</td>
<td>A string specifying the name of an event you define</td>
</tr>
<tr>
<td>triggerpercent</td>
<td>A long identifying the number of activities the BuildModel function should process before triggering the eventname event</td>
</tr>
</tbody>
</table>

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
- EventNotFound! – The event cannot be found on the passed progressobject, so the model cannot be built
**BuildModel**

- **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)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eventname</code></td>
<td>Name of the event</td>
</tr>
<tr>
<td><code>currentactivity</code></td>
<td>A long identifying the number of the current activity</td>
</tr>
<tr>
<td><code>totalnumberofactivities</code></td>
<td>A long identifying the total number of activities in the trace file</td>
</tr>
</tbody>
</table>

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

**Examples**

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

```
Profiling lpro_model
String ls_filename

lpro_model = CREATE Profiling
lpro_model.SetTraceFileName(ls_filename)
lpro_model.BuildModel()
```
This example creates a trace tree model of a trace file:

```powerscript
TraceTree ltct_model
String ls_filename

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

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

```powerscript
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**

Converts a number into a Byte datatype or obtains a Byte value stored in a blob.

### Syntax

```
Byte ( stringorblob )
```

### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringorblob</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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:

```
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:

```
Byte lb_byte
Blob myBlob
myBlob = Blob(sle_1.text)
lb_byte = Byte(myBlob)
```

### See also

GetByte
SetByte
CHAPTER 10  PowerScript Functions

Cancel
Description  Stops the execution of a pipeline object.
Applies to  Pipeline objects
Syntax  
\[
\text{pipelineobject}.\text{Cancel}() 
\]
Return value  Integer. Returns 1 if it succeeds and -1 if an error occurs.
Usage  Call this function only when Start or Repair is executing. When you stop a pipeline with Cancel, data is committed as if the pipeline had reached the maximum errors limit. You control how the pipeline behaves when it reaches the limit in the Data Pipeline painter (see the PowerBuilder User's Guide).
Examples  This statement for a CommandButton’s Clicked script allows the user to stop the execution of the pipeline \(i_{\text{pipe}}\):  
\[
i_{\text{pipe}}.\text{Cancel}() 
\]
See also  Repair

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  
\[
\text{editname}.\text{CanUndo}() 
\]
Argument  
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editname</td>
<td>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.</td>
</tr>
</tbody>
</table>
CategoryCount

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:

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

See also

Undo

CategoryCount

Description

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

Applies to

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

Syntax

`controlname.CategoryCount([ graphcontrol ])`

<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 number of categories, or the name of a DataWindow control containing the graph.</td>
</tr>
<tr>
<td><code>graphcontrol</code> (DataWindow control only) (optional)</td>
<td>A string whose value is the name of the graph in the DataWindow for which you want the number of categories. Graphcontrol is required if <code>controlname</code> is a DataWindow control.</td>
</tr>
</tbody>
</table>

Return value

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

Examples

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

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

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

```powerbuilder
integer li_count
li_count = gr_product_data.CategoryCount()
```
See also

DataCount
SeriesCount

CategoryName

Description
Obtains the category name associated with the specified category number.

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

Syntax
controlname.CategoryName ( { graphcontrol, } categorynumber )

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:

```power
defines 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:

```power
defines 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
\[ \text{Ceiling} (\ n) \]

<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 \( n \) to 5:
\[
\begin{align*}
\text{decimal dec, num} \\
\text{dec} = 4.8 \\
\text{num} = \text{Ceiling}(\text{dec})
\end{align*}
\]

These statements set \( n \) to −4:
\[
\begin{align*}
\text{decimal num} \\
\text{num} = \text{Ceiling}(-4.2) \\
\text{num} = \text{Ceiling}(-4.8)
\end{align*}
\]

See also
Int
Round
Truncate
Ceiling method for DataWindows in the DataWindow Reference or online Help

ChangeDirectory

Description
Changes the current directory.

Syntax
\[ \text{ChangeDirectory} (\ directoryname) \]

<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.
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:

```powerscript
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 )
```

#### Argument | Description
--- | ---
`windowname` | The name of the window for which you want to change the menu.
`menuname` | The name of the menu you want to make the current menu.
`position` (MDI frame windows only) | The number of the item on the menu bar to which you want to append the names of the open sheets. Items on the menu bar are numbered from the left, beginning with 1. The default is 0, which lists the open sheets on the menu bar’s next-to-last menu (or the last menu if there is only one available).

#### Return value

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

#### Usage

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

#### Examples

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

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

Description
Extracts the first 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:

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

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

```powerbuilder
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>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 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:

```powerscript
string ls_S
ls_S = CharA(42)
```

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

```powerscript
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
menuname.Check()

Argument | Description
--- | ---
menuname | The fully qualified name of the menu next to which you want to display a checkmark. The item must be in a 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 (optional)</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:

```power
global long red, green, blue
global long custom[1]
global 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 data type 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>
ClassList

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:

```powerbuilder
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**

```powerscript
controlname.ClassName()
```

<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 know the name assigned to the control in the style window (the class of the control)</td>
</tr>
</tbody>
</table>

**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`:

```powerscript
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:

```powerbuilder
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*):

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

See also

- DraggedObject
- TypeOf

### Syntax 2

**For variables**

**Description**

Provides the datatype of a variable.

**Syntax**

```
ClassName ( variable )
```

**Argument** | **Description**
--- | ---
`variable` | The name of the variable for which you want to know its name (that is, its datatype)

**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:

```powerbuilder
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
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.
ClearBoldDates

Examples

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

```powerbuilder
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:

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

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

```powerbuilder
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:

```powerbuilder
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

```powerbuilder
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`:

```powerscript
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**

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

**Description**

Retrieves or replaces the contents of the system clipboard with text. If `string` is specified, `Clipboard` returns the current contents and replaces it with `string`.

**Syntax**

```powerscript
Clipboard {{string}}
```

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

**Return value**

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

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

**Usage**

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

```powershell
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`:

```powershell
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:

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

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

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

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

```powershell
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
CHAPTER 10  PowerScript Functions

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>

(DataWindow control and DataStore only) (optional)

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>
Close

Syntax 1

**For windows**

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

**Description**

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

**Applies to**

Window objects

**Syntax**

```
Close (windowname)
```

**Argument** | **Description**
---|---
`windowname` | The name of the window you want to close

**Return value**

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

**Usage**

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

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

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

**Preventing a window from closing**

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

**Examples**

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

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

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

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

**See also**

Hide
Open

356

PowerBuilder
CHAPTER 10 PowerScript Functions

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
olestorage.Close ( )

Return value
Integer. Returns 0 if it succeeds and one of the following negative values if an error occurs:
-1 The storage is not open
-9 Other error

If olestorage is null, Close returns null.

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

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 ( )

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

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

-1  The storage is not open
-9  Other error

If olestream is null, Close returns null.

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

```plaintext
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  
```plaintext
instancename.Close()
```

<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  ErrorReturn. Returns one of the following values:

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

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

Examples  This example closes a trace file:

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

See also  Reset

Open

NextActivity
CHAPTER 10  PowerScript Functions

CloseChannel

Description
Closes a DDE channel.

Syntax

```powerbuilder
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:

```powerbuilder
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>tabcontrolname</td>
<td>The name of the Tab control containing the tab page you want to close</td>
</tr>
<tr>
<td>userobjectvar</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.

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

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

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

```
tab_personnel.CloseTab(u_employee)
tab_personnel.OpenTab(u_departments)
```
When the user chooses a menu item that closes a user object, the following excerpt from the menu item’s script prompts the user for confirmation before closing the u_employee user object in the window to which the menu is attached:

```
IF MessageBox("Close ", "Close?", &
   Question!, YesNo!) = 1 THEN
   // User chose Yes, close user object.
   ParentWindow.CloseTab(u_employee)
   // If user chose No, take no action.
END IF
```

See also: OpenTab

### CloseUserObject

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

**Applies to**
Window objects

**Syntax**
```
windowname.CloseUserObject(userobjectname)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowname</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`:

```powerbuilder
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:

```powerbuilder
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**

```powerbuilder
CloseWithReturn ( windowname, returnvalue )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windowname</code></td>
<td>The name of the window you want to close.</td>
</tr>
<tr>
<td><code>returnvalue</code></td>
<td>The value you want to store in the Message object when the window is closed.</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 \textit{CloseWithReturn} is to close a response window and return information from the response window to the window that opened it. Use \textit{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 \textit{Close}, \textit{CloseWithReturn} removes a window from view, closes it, and executes the script for the \textit{CloseQuery} and \textit{Close} events, if any. Before executing the event scripts, it also stores \textit{returnvalue} in the Message object. Then PowerBuilder executes the rest of the script that called the \textit{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 \textit{returnvalue} in the Message object properties according to its datatype. In the script that called \textit{CloseWithReturn}, you can access the returned value by specifying the property of the Message object that corresponds to the return value’s datatype.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|}
\hline
\textbf{Return value datatype} & \textbf{Message object property} \\
\hline
Numeric & Message.DoubleParm \\
PowerObject (such as a structure) & Message.PowerObjectParm \\
String & Message.StringParm \\
\hline
\end{tabular}
\caption{Message object properties where return values are stored}
\end{table}

Returning several values as a structure

To return several values, create a user-defined structure to hold the values and access the \textit{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 \textit{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 \textit{CloseQuery} event. Use a \textbf{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:

```powershell
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><code>treeviewname</code></td>
<td>The TreeView control in which you want to collapse an item</td>
</tr>
<tr>
<td><code>itemhandle</code></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 11_tvi
11_tvi = tv_list.FindItem(currenttreeitem!, 0)
tv_list.CollapseItem(11_tvi)
```

This example collapses the top-level item in a TreeView control that has only one level-one entry:
```
long 11_tvi
11_tvi = tv_list.FindItem(roottreeitem!, 0)
tv_list.CollapseItem(11_tvi)
```

**See also**
ExpandItem
ExpandAll
FindItem
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:

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

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

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

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

li_argcnt = 1
DO WHILE Len(ls_cmd) > 0
  // Find the first blank
```
i = Pos( ls_cmd, " ")

// If no blanks (only one argument),
// set i to point to the hypothetical character
// after the end of the string
if i = 0 then i = Len(ls_cmd) + 1

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

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

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

LOOP

CommitTransaction
Description
Declares that the EAServer transaction associated with the calling thread should be committed.

Applies to
CORBACurrent objects

Syntax
CORBACurrent.CommitTransaction (breportheuristics )

Argument | Description
--- | ---
CORBACurrent | Reference to the CORBACurrent service instance
breportheuristics | A boolean specifying whether heuristic decisions should be reported for the transaction associated with the calling thread

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
**CommitTransaction**

-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:

```powerbuilder
// 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
  // 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><code>oleobject</code></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><code>classname</code></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`.

**Examples**

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

```powershell
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:

```powershell
OLETxnObject EmpObj
Integer li_rc
EmpObj = CREATE OLETxnObject
li_rc = EmpObj.ConnectToNewObject("PB70COM.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`
CHAPTER 10  PowerScript Functions

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><code>oleobject</code></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><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>classname</code></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.
This example creates an OLEObject variable and calls ConnectToObject 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 } )
```

<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 <code>filename</code>, in which case you must specify <code>classname</code>. <code>Oleobject</code> is connected to the active object in the server application specified in <code>classname</code>.</td>
</tr>
<tr>
<td>classname  (optional)</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. If you omit <code>classname</code>, PowerBuilder uses the extension of <code>filename</code> 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 Invalid Call: this function not applicable to OLETxnObject

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`:

```powerbuilder
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:

```powerbuilder
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):

```powerbuilder
integer result
OLEObject myoleobject

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

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

```powerbuilder
integer result
OLEObject myoleobject

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

See also

ConnectToNewObject
DisconnectObject
ConnectToRemoteObject

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

Applies to
OLEObject objects

Syntax
`oleobject.ConnectToRemoteObject ( hostname, filename [, classname ] )`

<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
ConnectToRemoteObject

-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 DCOMCNFGEXE 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:

```powerbuilder
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:

```powerscript
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()`

<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 you want to use to establish the connection. The Connection object has properties that specify how the connection will be established.</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 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`:

```powerbuilder
// Global variable:
// connection myconnect
long ll_rc
myconnect = create connection
myconnect.driver = "jaguar"
myconnect.location = "Jagserver1:9000"
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:

```powerbuilder
SSLServiceProvider  sp
// set QOP
getcodecontextservice ( "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 ( RuntimeException 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: dwcontrol.Object.dwobjectname</td>
</tr>
<tr>
<td></td>
<td>If objectref is a DataWindow, text is copied from the edit control over the current row and column. If objectref is a DropDownListBox or DropDownPictureListBox, its AllowEdit property must be true.</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.

For OLE controls and OLE DWObjects, Copy returns 0 if it succeeds and one of the following negative values if an error occurs:
-1 Container is empty
-2 Copy Failed
-9 Other error

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

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

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

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

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

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

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

See also
Clear
Clipboard
Cut
Paste
ReplaceText
SelectText

---

**CopyRTF**

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

**Applies to**
DataWindow controls, DataStore objects, and RichTextEdit controls

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</td>
<td>The name of the DataWindow control, DataStore object, or RichTextEdit control from which you want to copy the selection in rich text format. The DataWindow object in the DataWindow control or DataStore must be a RichText DataWindow.</td>
</tr>
</tbody>
</table>
### 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`:

```powerbuilder
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`:

```powerbuilder
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
- A Cos
- 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:

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

// Set the start equal to the current CPU usage.
ll_start = Cpu()
...
// Executable statements being timed

// Set ll_used to the number of CPU seconds that were used (current CPU time - start).
ll_used = Cpu() - ll_start
```

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:

```plaintext
string ls_path="my targets"
integer li_filenum
CreateDirectory ( ls_path )
li_filenum = ChangeDirectory( ls_path )
```

See also
GetCurrentDirectory
RemoveDirectory
CHAPTER 10  PowerScript Functions

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>From a PowerBuilder client</td>
</tr>
<tr>
<td>Syntax 1</td>
</tr>
<tr>
<td>From within an EAServer or COM+ component</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

\[\text{connection}.\text{CreateInstance} (\text{objectvariable} [, \text{classname}])\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{connection}</td>
<td>The name of the Connection object used to establish the connection.</td>
</tr>
<tr>
<td>\text{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>\text{classname} (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, “mypkg/mymodule”).</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:

```powerbuilder
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
CHAPTER 10  PowerScript Functions

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  
\[\text{transactionserver}.\text{CreateInstance}(\text{objectvariable}, \text{classname})\]

<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>
<tr>
<td>classname (optional)</td>
<td>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, &quot;mypackage/mycomponent&quot;). For COM+ components, you can optionally prepend a ProgID followed by a period to the class name (for example, &quot;PowerBuilder:HTMLDataWindow&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  The CreateInstance function on the TransactionServer context object allows you to access other EAServer or COM+ components running on the current server.

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.

The following statements show how an EAServer component might instantiate another component in the same server and call one of its methods:

```powerbuilder
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:

```powerbuilder
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")
IF rc <> 0 THEN
    return "Error from CreateInstance " + String (rc)
END IF
iole.my_func ()
```
CreatePage

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

Applies to
User objects used as tab pages

Syntax
userobject.CreatePage()

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

Return value
Integer. Returns one of the following values: 1 if the page is successfully created and -1 if the page was already created or if it 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:

```
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
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 DropDownListListBox 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.

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:

```plaintext
mle_emp_address.Cut()
```

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

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

**See also**

Copy
Clear
Clipboard
DeleteItem
Paste
**DataCount**

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

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

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

<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
DataSources

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
\texttt{rtename.DataSource ( dwsource )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{rtename}</td>
<td>The name of the RichTextEdit control for which you want to get data in a DataWindow</td>
</tr>
<tr>
<td>\texttt{dwsource}</td>
<td>The name of the DataWindow control, DataStore, or child DataWindow that contains the data to be connected with input fields in \texttt{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
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</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>in a blob</td>
<td></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 )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| datetime | A DateTime value or a blob in which the first value is a date or
|           | DateTime value. The rest of the contents of the blob is ignored.
|           | Datetime can also be an Any variable containing a DateTime or
|           | blob. |

Return value
Date. Returns the date in datetime as a date. If datetime contains an invalid date
or an incompatible datatype, Date returns 1900-01-01. If datetime is null, Date
returns null.
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`:

```powerbuilder
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:

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

### Syntax 2

**For strings**

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

**Date ( string )**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
<td>A string containing a valid date (such as January 1, 2002, or 12-31-99) that you want returned as a date. <code>DateTime</code> 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`.

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:

```
Date("2004/07/04")
Date("2004 July 4")
Date("04 July 2004")
```

**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 uses the `IsDate` function to check the text before using `Date` to convert it:

**Version 1:**

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

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

**Version 2:**

```
date ld_my_date

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

**See also**

- `DateTime`
- `IsDate`
- `RelativeDate`
- `RelativeTime`
- `Date` 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
\[
\text{Date}(\text{year}, \text{month}, \text{day})
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>The 4-digit year (-9999 to 9999) of the date</td>
</tr>
<tr>
<td>month</td>
<td>The 1- or 2-digit integer for the month (1 to 12) of the year</td>
</tr>
<tr>
<td>day</td>
<td>The 1- or 2-digit integer for the day (1 to 31) of the month</td>
</tr>
</tbody>
</table>

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:

```powershell
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>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<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>
CHAPTER 10  PowerScript Functions

Syntax 1  For creating DateTime values

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

Syntax

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>A value of type date.</td>
</tr>
<tr>
<td>time</td>
<td>A value of type time. If you omit time, PowerBuilder sets time to 00:00:00.000000 (midnight). If you specify time, only the hour portion is required.</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:

```
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>
Day

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):

```powershell
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:

```powershell
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:

```powershell
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
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
```
### DayNumber

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

**Syntax**

```powerbuilder
DayNumber (date)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>The date value from which you want the number of the day of the week</td>
</tr>
</tbody>
</table>

**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):

```powerbuilder
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 the `sle_date`:

```powerbuilder
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
DayNumber
IsDate
DayName in the *DataWindow Reference*
ELSE
    MessageBox("Error", &
    "This date is invalid: " &
    + sle_date.Text)
END IF

See also
Day
DayName
IsDate
DayNumber in the DataWindow Reference

DaysAfter

Description
Determines the number of days one date occurs after another.

Syntax
DaysAfter (date1, date2)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date1</td>
<td>A date value that is the start date of the interval being measured</td>
</tr>
<tr>
<td>date2</td>
<td>A date value that is the end date of the interval</td>
</tr>
</tbody>
</table>

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:

DaysAfter(2002-12-24, 2002-12-20)

This statement returns 0:

DaysAfter(2003-12-24, 2003-12-24)

This statement returns 5:

DaysAfter(2003-12-29, 2004-01-03)

If you declare date1 and date2 date variables and assign February 16, 2003, to date1 and April 28, 2003, to date2 as follows:

date date1, date2
DBHandle

Description
Reports the handle for your DBMS.

Applies to
Transaction objects

Syntax
`transactionobject.DBHandle()`

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.

\[\text{date1} = 2003-02-16\]
\[\text{date2} = 2003-04-28\]

then each of the following statements returns 71:

\[\text{DaysAfter(date1, date2)}\]
\[\text{DaysAfter}(2003-02-16, \text{date2})\]
\[\text{DaysAfter}(\text{date1}, 2003-04-28)\]
\[\text{DaysAfter}(2003-02-16, 2003-04-28)\]
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:

```PowerScript
IF IsNull(auo_ext) THEN DebugBreak()
```

Dec

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

Syntax
`Dec(stringorblob)`

<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 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. <code>Stringorblob</code> can also be an <code>Any</code> variable containing a string or blob.</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)
```
DeleteCategory

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

See also

Double
Integer
Long
Real

DeleteCategory

Description

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

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

Syntax

`controlname.DeleteCategory (categoryvalue)`

<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 <code>controlname</code>. The value you specify must be the same datatype as the datatype of the category axis.</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, DeleteCategory returns null.

Examples

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

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

See also

DeleteData
DeleteSeries
### DeleteColumn

**Description**
Deletes a column.

**ListView controls**

**Syntax**

```plaintext
listviewname.DeleteColumn(index)
```

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

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

**Examples**
This example deletes the second column in a ListView control:

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

**See also**
DeleteColumns

---

### DeleteColumns

**Description**
Deletes all columns.

**ListView controls**

**Syntax**

```plaintext
listviewname.DeleteColumns()
```

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

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

**Examples**
This example deletes all columns in a ListView control:

```plaintext
lv_list.DeleteColumns()
```

**See also**
DeleteColumn
DeleteData

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

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

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

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

**Return value**
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, DeleteData returns null.

**Examples**
These statements delete the data at data point 7 in the series named Costs in the graph *gr_product_data*:
```
integer SeriesNbr
// Get the number of the series.
SeriesNbr = gr_product_data.FindSeries("Costs")
gr_product_data.DeleteData( SeriesNbr, 7 )
```

**See also**
AddData
DeleteCategory
DeleteSeries
FindSeries

DeleteItem

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

<table>
<thead>
<tr>
<th>To delete an item from</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ListBox or DropDownListBox control</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>A ListView control</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>A TreeView control</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>
CHAPTER 10  PowerScript Functions

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)

Arguments  
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listboxname</td>
<td>The name of the ListBox, DropDownListBox, PictureListBox, or DropDownPictureListBox from which you want to delete an item</td>
</tr>
<tr>
<td>index</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)

These statements delete the first selected item in lb_actions:

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

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

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

See also  
AddItem
FindItem
InsertItem
SelectItem
DeleteItem

Syntax 2 For ListView controls
Description
Deletes the specified item from a ListView control.
Applies to
ListView controls
Syntax
`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:

```plaintext
integer index
index = lv_list.selectedIndex()
lv_list.DeleteItem(index)
```

See also
AddItem
FindItem
InsertItem
SelectItem
DeleteItems

Syntax 3 For TreeView controls
Description
Deletes an item from a control and all its child items, if any.
Applies to
TreeView controls
Syntax
`treeviewname.DeleteItem ( itemhandle )`

<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 RootTreeWidgetItem 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
InsertItem
SelectItem
DeleteItems

---

**DeleteItems**

**Description**

Deletes all items from a ListView control.

**Applies to**

ListView controls

**Syntax**

```powerScript
listviewname.DeleteItems()
```

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

**Return value**

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

**Examples**

This example deletes all the items in a ListView control:

```powerScript
lv_list.DeleteItems()
```

See also

DeleteItem
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><code>listviewname</code></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><code>index</code></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><code>listviewname</code></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>

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
CHAPTER 10 PowerScript Functions

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()

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The control in which you want to delete all pictures from the image list</td>
</tr>
</tbody>
</table>

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

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

tv_list.DeletePictures()

See also
AddPicture
DeletePicture
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>controlname</td>
<td>The graph in which you want to delete a series</td>
</tr>
<tr>
<td>seriesname</td>
<td>A string whose value is the name of the series you want to delete from controlname</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><code>listviewname</code></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><code>index</code></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:

```
lv_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><code>listviewname</code></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:

```
lv_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)}

Return value
\texttt{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()}

Return value
\texttt{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  \textit{instancename}.\texttt{DestroyModel} ( )

Return value  
ErrorReturn. Returns one of the following values:
\begin{itemize}
  \item Success! – The function succeeded
  \item ModelNotExistsError! – The function failed because no model exists
\end{itemize}

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:
\begin{verbatim}
  lpro_model.DestructModel()
  DESTROY lpro_model
\end{verbatim}

See also  BuildModel

DirectoryExists

Description  Determines if the named directory exists.
Syntax  DirectoryExists ( \texttt{directoryname} )

Return value  Returns \texttt{true} if the directory exists. Returns \texttt{false} if the directory does not exist or if you pass a file name in the \texttt{directoryname} argument.

Usage  You can use this method before attempting to move a file or delete a directory using other file system methods.
DirList

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>
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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filetype</td>
<td>An unsigned integer representing one or more types of files you want to list in the ListBox. Types are:</td>
</tr>
<tr>
<td></td>
<td>• 0 – Read/write files</td>
</tr>
<tr>
<td></td>
<td>• 1 – Read-only files</td>
</tr>
<tr>
<td></td>
<td>• 2 – Hidden files</td>
</tr>
<tr>
<td></td>
<td>• 4 – System files</td>
</tr>
<tr>
<td></td>
<td>• 16 – Subdirectories</td>
</tr>
<tr>
<td></td>
<td>• 32 – Archive (modified) files</td>
</tr>
<tr>
<td></td>
<td>• 16384 – Drives</td>
</tr>
<tr>
<td></td>
<td>• 32768 – Exclude read/write files from the list</td>
</tr>
<tr>
<td></td>
<td>To list several types, add the numbers associated with the types. For example, to list read-write files, subdirectories, and drives, use 0+16+16384 or 16400 for filetype.</td>
</tr>
<tr>
<td>statictext</td>
<td>The name of the StaticText in which you want to display the current drive and directory.</td>
</tr>
</tbody>
</table>
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:

```powershell
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:

```powershell
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:

```powershell
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 DropDownListPictureListBox controls

**Syntax**

```powershell
listboxname.DirSelect ( selection )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listboxname</code></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><code>selection</code></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.
Chapter 10  PowerScript Functions

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:

```
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><code>menuname</code></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:
```powerbuilder
m_appl.m_edit.Enabled = FALSE
```
is equivalent to:
```powerbuilder
m_appl.m_edit.Disable()
```

**Examples**
This statement disables the `m_edit` menu item on the menu `m_appl`:
```powerbuilder
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 **
```powerbuilder
transactionserver.DisableCommit()
```

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

```powershell
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**

```
Double ( stringorblob )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringorblob</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. Stringorblob 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:

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

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

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

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

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

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

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:

integer result
result = ole_1.DoVerb(7)
This example executes verb 7 for the object in the OLE DObject `ole_graph`:

```power
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>
</tbody>
</table>
| `dragmode` | A value of the DragMode datatype indicating the action you want to take on control:  
  - Begin! – Put `control` in drag mode  
  - Cancel! – Stop dragging `control` but do not cause a DragDrop event  
  - End! – Stop dragging `control` and if `control` is over a target object, cause a DragDrop event |

**Return value**

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

For OLE controls, returns the following values:

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

If any argument’s value is null, `Drag` returns `null`.

**Usage**

To see the list of draggable controls, open the Browser. All the objects in the hierarchy below `dragobject` are draggable.
If you set the control’s DragAuto property to `true`, PowerBuilder automatically puts the control in drag mode when the user clicks it. The user must hold the mouse button down to drag.

When you use `Drag(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):

```powerbuilder
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:

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

### See also

`DraggedObject`
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:

```power_script
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.

### 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)`

**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 1

### To enable editing of a label in a

<table>
<thead>
<tr>
<th>ListView control</th>
<th>Syntax 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeView control</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>
Enable

**Syntax 2**

For editing a label in a TreeView

**Description**

Puts a label in a TreeView into edit mode.

**Applies to**

TreeView controls

**Syntax**

```
treeviewname.EditLabel ( itemhandle )
```

**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:

```powershell
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 ( )
```

**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()
```

**Arguments**

<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 `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
```
EntryList

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
DisableCommit
IsInTransaction
IsTransactionAborted
Lookup
SetAbort
SetComplete
Which

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.

### Syntax

<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.
**ExecRemote**

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>
<tr>
<td><code>applname</code></td>
<td>A string whose value is the DDE name of the server application.</td>
</tr>
<tr>
<td><code>topicname</code></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):

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

See also

- `CloseChannel`
- `GetRemote`
- `OpenChannel`
- `SetRemote`
- `StartHotLink`
### 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><code>command</code></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><code>handle</code></td>
<td>A long that identifies the channel to the DDE server application. The <code>OpenChannel</code> function returns <code>handle</code> when you call it to open a DDE channel.</td>
</tr>
<tr>
<td><code>windowhandle</code> (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 <code>windowhandle</code>, 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 to 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))
```

**See also**
- CloseChannel
- GetRemote
- OpenChannel
- SetRemote

---

### Exp

**Description**

Raises \(e\) to the specified power.

\[\text{Exp}(n)\]

**Syntax**

<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, \(\text{Exp}\) returns null.

**Inverse of Exp**

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

**Examples**

This statement returns 7.38905609893065.

\[\text{Exp}(2)\]

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 = \text{Exp}(x)
```

**See also**
- Log
- LogTen
- `Exp` method for DataWindows in the *DataWindow Reference* or online Help.
CHAPTER 10  PowerScript Functions

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>treeviewname</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>itemhandle</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>treeviewname</td>
<td>The TreeView control in which you want to expand an item</td>
</tr>
<tr>
<td>itemhandle</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.
Fact

Usage
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:

```powershell
long 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
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 * 2 * 3 * 4):

```powershell
Fact(4)
```

Both these statements return 1:

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

See also
Fact method for DataWindows in the DataWindow Reference or online Help
CHAPTER 10  PowerScript Functions

FileClose

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

Syntax
FileClose ( file# )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file#</td>
<td>The integer assigned to the file you want to close. The FileOpen function returns the file number when it opens the file.</td>
</tr>
</tbody>
</table>

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:

```power"
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 } )

<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 copy</td>
</tr>
<tr>
<td>targetfile</td>
<td>String for the name of the file you are copying to</td>
</tr>
<tr>
<td>replace (optional)</td>
<td>Boolean specifying whether to replace the target file (true) or not (false)</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns values as follows:

- 1 – Success
- -1 – Error opening sourcefile
- -2 – Error writing targetfile
**FileDelete**

**Usage**

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:

```plaintext
integer li_FileNum
li_FileNum = FileCopy ("jazz.gif", & "C:\emusic\jazz.gif", FALSE)
```

**See also**

FileMove
GetCurrentDirectory

---

**FileDelete**

**Description**

Deletes the named file.

**Syntax**

```
FileDelete ( filename )
```

**Argument** | **Description**
--- | ---
filename | A string whose value is the name of the file you want to delete

**Return value**

Boolean. Returns `true` if it succeeds, `false` if an error occurs. If `filename` is `null`, `FileDelete` returns `null`.

**Examples**

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

```plaintext
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)
```

**Argument** | **Description**
--- | ---
filename | The name of the file you want to test for encoding type

**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:

```powerscript
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)
```
**FileExists**

**Description**

Reports whether the specified file exists.

**Syntax**

FileExists ( filename )

**Argument** | **Description**  
---|---  
filename | A string whose value is the name of a file

**Return value**

Boolean. Returns true if the file exists, false if it does not exist. If filename is null, FileExists returns null.

**Usage**

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

**Examples**

This example determines if the file the user selected in the Save File window exists and, if so, asks the user if the file can be overwritten:

```powerbuilder
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

---

See also

Blob
FileClose
FileOpen
FileReadEx
FileWriteEx
String
**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**

```plaintext
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:

```plaintext
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:

```plaintext
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**

```plaintext
FileOpen ( filename {, filemode {, fileaccess {, filelock {, writemode { encoding ]}]}]} )
```

<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 file you want to open. If <code>filename</code> is not on the current directory’s relative search path, you must enter the fully qualified name.</td>
</tr>
<tr>
<td><code>filemode</code></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><code>fileaccess</code></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><code>filelock</code></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><code>writemode</code></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><code>Writemode</code> is ignored if the fileaccess argument is Read!</td>
</tr>
</tbody>
</table>
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>
<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), linefeed (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:

```powerscript
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 user can write to the file (LockWrite!):

```powerscript
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:

```powerscript
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
FileLength64
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.

Syntax
FileRead ( file#, variable )

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

Return value
Integer. Returns the number of 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:

```powerscript
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:

```powerscript
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 )

Argument Description

<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.

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.

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:

```c
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.
CHAPTER 10  PowerScript Functions

Examples

This example positions the file pointer 14 bytes from the end of the file:

```
integer li_FileNum
li_FileNum = FileOpen("emp_data")
FileSeek(li_FileNum, -14, FromEnd!)
```

This example moves the file pointer from its current position 14 bytes toward the end of the file. In this case, if no processing has occurred after FileOpen to affect the file pointer, specifying FromCurrent! is the same as specifying FromBeginning!:

```
integer li_FileNum
li_FileNum = FileOpen("emp_data")
FileSeek(li_FileNum, 14, FromCurrent!)
```

See also

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

```
FileSeek64 ( file#, position, origin )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>file#</code></td>
<td>The integer assigned to the file when it was opened.</td>
</tr>
<tr>
<td><code>position</code></td>
<td>A long whose value is the new position of the file pointer relative to the position specified in <code>origin</code>, in bytes.</td>
</tr>
<tr>
<td><code>origin</code></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 68 00 69 00 73 00`, the following example illustrates the behavior:

```c
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:

```c
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!`:

```c
integer li_FileNum
li_FileNum = FileOpen("emp_data")
FileSeek64(li_FileNum, 14, FromCurrent!)
```

### See also

- FileReadEx
- FileSeek
- FileWriterEx
CHAPTER 10  PowerScript Functions

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
FileWrite ( file#, variable )

Argument  Description
file#      The integer assigned to the file when the file was opened
variable  A string or blob whose value is the data you want to write to the file

Return value
Integer. Returns the number of 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:

```powerbuilder
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:

    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:

    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
CHAPTER 10  PowerScript Functions

**Fill**

**Description**
Builds a string of the specified length by repeating the specified characters until the result string is long enough.

**Syntax**
```
Fill ( chars, n )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chars</code></td>
<td>A string whose value will be repeated to fill the return string</td>
</tr>
<tr>
<td><code>n</code></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)
```

**See also**
Space
*Fill* method for DataWindows in the *DataWindow Reference* or online Help
### FillA

**Description**
Builds a string of the specified length in bytes by repeating the specified characters until the result string is long enough.

**Syntax**
```powerbuilder
FillA(chars, n)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chars</td>
<td>The string whose value is repeated to fill the return string</td>
</tr>
<tr>
<td>n</td>
<td>A long specifying the number of bytes in the return string</td>
</tr>
</tbody>
</table>

**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**
```powerbuilder
FillW(chars, n)
```

### Find

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

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.</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 cursor 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 cursor 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 forward is true or at the end if forward 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:

    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**
```
controlname.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.

**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`:
```
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`:
```
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}
```

**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`:
```powershell
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`:
```powershell
ClassDefinition cd_windef
string ls_libraries[ ]

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

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

**See also**
- FindFunctionDefinition
- FindMatchingFunction
- FindTypeDefinition
FindFunctionDefinition

Description

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

Syntax

FindFunctionDefinition (functionname[, librarylist])

<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.

Examples

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

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

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

```powershell
ScriptDefinition sd_myfunc
string ls_libraries[ ]

ls_libraries[1] = "c:\pwr_app\windows.pbl"
ls_libraries[2] = "c:\pwr_app\framework\windows.pbl"
ls_libraries[3] = "c:\pwr_app\framework\ancestor.pbl"

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

See also

FindClassDefinition
FindMatchingFunction
FindTypeDefinition
FindItem

Finds the next item in a list.

To find the next item

| In a ListBox, DropDownListBox, PictureListBox, or DropDownPictureListBox | Syntax 1 |
| In a ListView control based upon its label | Syntax 2 |
| By relative position in a ListView control | Syntax 3 |
| By relative position in a TreeView control | Syntax 4 |

Syntax

For ListBox and DropDownListBox controls

Description

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

Applies to

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

`listboxname.FindItem( text, index )`

Argument | Description
--- | ---
`listboxname` | The name of the ListBox control in which you want to find an item.
`text` | A string whose value is the starting text of the item you want to find.
`index` | The number of the item just before the first item to be searched. To search the whole list, specify 0.

Return value

Integer. Returns the index of the first matching item. To match, the item must start with the specified text; however, the text in the item can be longer than the specified text. If no match is found or if an error occurs, `FindItem` returns -1. If any argument’s value is null, `FindItem` returns null.

Usage

When `FindItem` finds the matching item, it returns the index of the item but does not select (highlight) the item. To find and select the item, use the `SelectItem` function.

Examples

Assume the ListBox `lb_actions` contains the following list:

<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). \texttt{FindItem} sets \texttt{Index} to 4:

\begin{verbatim}
integer Index
Index = lb_actions.FindItem("Delete", 1)
\end{verbatim}

See also
AddItem
DeleteItem
InsertItem
SelectItem

### Syntax 2

**For ListView controls**

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

**Applies to**

ListView controls

**Syntax**

\begin{verbatim}
listviewname.FindItem ( startindex, label, partial, wrap )
\end{verbatim}

<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>label</td>
<td>The string that is the target of the search</td>
</tr>
<tr>
<td>partial</td>
<td>If set to \texttt{true}, the search looks for a partial label match</td>
</tr>
<tr>
<td>wrap</td>
<td>If set to \texttt{true}, the search returns to the first index item after it has finished</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 \texttt{startindex + 1} by default. To search from the beginning, specify 0.

If \texttt{partial} is set to \texttt{true}, the search string matches any label that begins with the specified text. If \texttt{partial} is set to \texttt{false}, the search string must match the entire label.

If \texttt{wrap} is set to \texttt{true}, the search wraps around to the first index item after searching to the end. If \texttt{wrap} is set to \texttt{false}, the search stops at the last index item in the ListView.

\texttt{FindItem} does not select the item it finds. You must use the item’s selected property in conjunction with \texttt{FindItem} to select the resulting match.
### Examples

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

```power_script
listviewitem l_lvi
integer li_index
string ls_label

ls_label = sle_find.Text
IF ls_label = "" THEN
    MessageBox("Error", &
    "Enter the name of a list item")
    sle_find.SetFocus()
ELSE
    li_index = lv_list.FindItem(0, ls_label, TRUE, TRUE)
END IF
IF li_index = -1 THEN
    MessageBox("Error", "Item not found.")
ELSE
    lv_list.GetItem(li_index, l_lvi)
    l_lvi.HasFocus = TRUE
    l_lvi.Selected = TRUE
    lv_list.SetItem(li_index, l_lvi)
END IF
```

### See also

- AddItem
- DeleteItem
- InsertItem
- SelectItem

### Syntax 3

#### For ListView controls

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

**Description**

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

**Applies to**

ListView controls

**Syntax**

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listviewname</code></td>
<td>The ListView control for which you want to search for items.</td>
</tr>
<tr>
<td><code>startindex</code></td>
<td>The index number from which you want your search to begin.</td>
</tr>
</tbody>
</table>
FindItem

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 the 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.

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:

```powerbuilder
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
```
For TreeView controls

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

Applies to
TreeView controls

Syntax
`treeviewname.FindItem (navigationcode, itemhandle)`

Arguments

<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 find a specified item.</td>
</tr>
<tr>
<td><code>navigationcode</code></td>
<td>A value of the TreeNavigation enumerated datatype specifying the relationship between <code>itemhandle</code> and the item you want to find. See the table in Usage note for a list of valid values.</td>
</tr>
<tr>
<td><code>itemhandle</code></td>
<td>A long for the handle of an item related via <code>navigationcode</code> to the item for which you are searching.</td>
</tr>
</tbody>
</table>

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 <code>itemhandle</code>. A sibling is an item at the same level with the same parent. Returns -1 if there are no more siblings.</td>
</tr>
</tbody>
</table>
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)
```

### Examples

<table>
<thead>
<tr>
<th>Navigation code value</th>
<th>What FindItem finds</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreviousTreeItem!</td>
<td>The sibling before <code>itemhandle</code>. Returns -1 if there are no more siblings.</td>
</tr>
<tr>
<td>ParentTreeItem!</td>
<td>The parent of <code>itemhandle</code>. Returns -1 if the item is at level 1.</td>
</tr>
<tr>
<td>ChildTreeItem!</td>
<td>The first child of <code>itemhandle</code>. 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 <code>itemhandle</code>, 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 <code>FindItem</code> and then <code>SelectItem</code>.</td>
</tr>
<tr>
<td>PreviousVisibleTreeItem!</td>
<td>The next expanded item before <code>itemhandle</code>, regardless of level. Returns -1 if the item is the first root item.</td>
</tr>
<tr>
<td>CurrentTreeItem!</td>
<td>The selected item. Returns -1 if the control never had focus and nothing has been selected.</td>
</tr>
<tr>
<td>DropHighlightedTreeItem!</td>
<td>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 <code>false</code> because of other activity in the control.</td>
</tr>
</tbody>
</table>

### 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  

\[
\text{classdefobject.FindMatchingFunction}(\text{scriptname}, \text{argumentlist})
\]

Return value  ScriptDefinition. Returns an object instance with information about the matching function. If no matching function is found, \text{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.

\textbf{Arguments passed by reference}  To find a function with an argument that is passed by reference, you must specify the \texttt{REF} keyword. If you have a \texttt{VariableDefinition} object for a function argument, check the \texttt{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.

### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{classdefobject}</td>
<td>The name of the ClassDefinition object describing the class in which you want to find a function.</td>
</tr>
<tr>
<td>\texttt{scriptname}</td>
<td>A string whose value is the name of the function.</td>
</tr>
<tr>
<td>\texttt{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:</td>
</tr>
<tr>
<td></td>
<td>{ ref } datatype { [] }</td>
</tr>
<tr>
<td></td>
<td>For a bounded array, the argument must include the range, as in:</td>
</tr>
<tr>
<td></td>
<td>ref integer[1 TO 10]</td>
</tr>
</tbody>
</table>
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:

```powershell
string ls_args[]
ScriptDefinition sd

ls_args[1] = "ref dragobject"
ls_args[2] = "double"
ls_args[3] = "integer"
ls_args[4] = "integer"

sd = c_obj.FindMatchingFunction( &
  "OpenUserObjectWithParm", ls_args)
IF NOT IsValid(sd) THEN
  mle_1.Text = "No matching script"
ELSE
  mle_1.Text = uf_scriptinfo(sd)
END IF
```

The `uf_scriptinfo` function gets information about the function that matched the signature and builds a string. `Scriptobj` is the ScriptDefinition object passed to the function:

```powershell
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

```
c ControlName.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 rte_1 for text the user specifies in the SingleLineEdit sle_search. The search proceeds forward from the cursor position, is case insensitive, and is not limited to whole words:

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

A second button labeled FindNext would have a script like this:

```
rte_1.FindNext ( )
```

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>seriesname</td>
<td>A string whose value is the name of the series for which you want the number</td>
</tr>
</tbody>
</table>

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 sleseries in SeriesNbr:

```pb
integer SeriesNbr
SeriesNbr = &
gr_product_data.FindSeries(sleseries.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:

```pb
integer SeriesNbr
SeriesNbr = &
dw_equipment.FindSeries("gr_computers", "PCs")
```

See also
AddSeries
DeleteSeries
FindCategory
CHAPTER 10  PowerScript Functions

FindTypeDefinition

Description
Searches for a type in one or more PowerBuilder libraries (PBLs) and provides information about its type definition. You can also get type definitions for system types.

Syntax
\[
\text{FindTypeDefinition} ( \text{typename} [, \text{librarylist}] )
\]

Argument | Description
--- | ---
\text{typename} | The name of a simple datatype, enumerated datatype, or class for which you want information. To find a type definition for a nested type, use this form:
\[
\text{libraryEntryName/typename}
\]
\text{librarylist} (optional) | An array of strings whose values are the fully qualified pathnames of PBLs. If you omit \text{librarylist}, \text{FindTypeDefinition} searches the library list associated with the running application.
PowerBuilder also searches its own libraries for built-in definitions, such as enumerated datatypes and system classes.

Return value
TypeDefinition. Returns an object reference with information about the definition of \text{typename}. If any arguments are null, \text{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 \text{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 \text{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:

\[
\begin{align*}
\text{TypeDefinition \ td_graphtype} \\
\text{EnumerationDefinition \ ed_graphtype} \\
\text{string \ enumname}
\end{align*}
\]
FindTypeDefinition

```powerbuilder
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:

```powerbuilder
TypeDefinition td_def
SimpleTypeDefinition std_def
EnumerationDefinition ed_def
ClassDefinition cd_def

td_def = FindTypeDefinition(typename)
CHOOSE CASE td_def.Category
CASE SimpleType!
    std_def = td_def
CASE EnumeratedType!
    ed_def = td_def
CASE ClassOrStructureType!
    cd_def = td_def
END CHOOSE
```

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

```powerbuilder
TypeDefinition 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)

Argument | Description
---|---
blob | A blob containing an ANSI character string you want to convert to a Unicode string

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.

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
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)

Argument | Description
---|---
blob | A blob containing a Unicode character string you want to convert to a string in the file format of the current version of PowerBuilder

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.
**GarbageCollect**

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:

```powerscript
long CollectTime

CollectTime = GarbageCollectGetTimeLimit()
```

See also

GarbageCollect
GarbageCollectSetTimeLimit

---

**GarbageCollectSetTimeLimit**

**Description**

Sets the minimum interval between garbage collection passes.

**Syntax**

```powerscript
GarbageCollectSetTimeLimit(newtimeinmilliseconds)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>newtimeinmilliseconds</code></td>
<td>A long (in milliseconds) that you want to set as the minimum period between garbage collection cycles.</td>
</tr>
<tr>
<td></td>
<td>If <code>null</code>, the existing interval is not changed.</td>
</tr>
</tbody>
</table>

**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:

```powershell
// Declare variable for active sheet
window activesheet
string mtext

activesheet = w_frame.GetActiveSheet()
IF IsValid(activesheet) THEN
   // There is an active sheet, so get its title;
   // change the text of the menu to read
   // Close plus the title of the active sheet
   mtext = "Close " + activesheet.Title
   m_main.m_file.m_close.Text = mtext
ELSE
   // No sheet is active, menu says Close Window
   m_main.m_file.m_close.Text = "Close Window"
END IF
```

See also
IsValid
GetAlignment

Description
Obtains the alignment of the paragraph containing the insertion point in a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
rename.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:

```powerscript
alignment l_align
l_align = rte_1.GetAlignment ()
```

See also
GetSpacing
GetTextStyle
SetAlignment
SetSpacing
SetTextStyle

GetApplication

Description
Gets the handle of the current Application object so you can get and set properties of the application.

Syntax
GetApplication ()

Return value
Application. Returns the handle of the current application object.

Usage
The GetApplication function lets you write generic code for an application, making it reusable in other applications. You do not have to code the actual name of the application when you want to set application properties.
To change whether Toolbar Tips are displayed, you can get the handle of the application object and set the ToolbarTips property:

```powerbuilder
application app
app = GetApplication()
app.ToolbarTips = FALSE
```

The previous example could be coded more simply as follows:

```powerbuilder
GetApplication().ToolbarTips = FALSE
```

---

### GetArgElement

**Description**

Returns the value in the specified argument.

**Applies to**

Window ActiveX controls

**Syntax**

```powerbuilder
activexcontrol.GetArgElement(index)
```

<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>

**Return value**

Any. Returns the specified argument.

**Usage**

Call this function after calling `InvokePBFunction` or `TriggerPBEvent` to access the updated value in an argument passed by reference.

JavaScript scripts must use this function to access arguments passed by reference. VBScript scripts can use this function if they established the argument list via calls to `SetArgElement`.

**Examples**

This JavaScript example calls the `GetArgElement` function:

```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);
...
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 )
```

**Argument** | **Description**
--- | ---
blobvariable | A variable of the Blob datatype from which you want to extract a value of the Byte datatype
n | The number of the position in blobvariable at which you want to retrieve a value of the Byte datatype
b | Variable of the Byte datatype in which you want to store the returned data of type Byte

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:

```power_script
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))
```

See also

Byte
SetByte

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
```
sslcallback.GetCertificateLabel ( thesessioninfo, labels )
```

<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>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>

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`:

```powerbuilder
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", &
        "Host: ", &
        sLocation + ":" + &
        "Port: ", &
        mySessionInfo.getProperty("port") + ":")
    CloseWithReturn("")
    IF numLabels > 0 THEN
        MessageBox ("Certificate labels:", &
            "Available certificates:", &
            "Selected certificate:", &
            "Select certificate:"
        )
        w_ssl_response.setVariable("Label", 
            w_ssl_response.getServerLocalObject().
            getCertificateLabel(idx))
        w_ssl_response.setVariable("Text", 
            w_ssl_response.getServerLocalObject().
            getCertificateText(idx))
        w_ssl_response.setVariable("Location", 
            w_ssl_response.getServerLocalObject().
            getCertificateLocation(idx))
        w_ssl_response.setVariable("LocationText", 
            w_ssl_response.getServerLocalObject().
            getCertificateLocationText(idx))
        IF w_ssl_response.getServerLocalObject().
            canAuthenticateCertificate(idx) THEN
            MessageBox ("Certificate accepted", &
                "Selected certificate: ", &
                w_ssl_response.getServerLocalObject().
                getCertificateLabel(idx)
            )
            CloseWithReturn(w_ssl_response.getServerLocalObject().
                getCertificateText(idx))
        ELSE
            MessageBox ("Certificate refused", &
                "Selected certificate: ", &
                w_ssl_response.getServerLocalObject().
                getCertificateLabel(idx)
            )
        END

    END

    CloseWithReturn("")
```

```powerbuilder```
"A certificate is required for connection to " + sLocation + "-nNo certificates are available")
ls_rc = ""
ELSE
sText = "Available certificates: "
FOR idx=1 to numLabels
sText += "-nCertificate[" + 
   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.

Applies to
TraceTreeObject, TraceTreeRoutine, and TraceTreeGarbageCollect objects

Syntax
instancename.GetChildrenList( list )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instancename</td>
<td>Instance name of the TraceTreeObject, TraceTreeRoutine, or TraceTreeGarbageCollect object.</td>
</tr>
<tr>
<td>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>
GetChildrenList

Return value

ErrorReturn. Returns the following values:

- **Success!** – The function succeeded
- **ModelError** – 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:

```powershell
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.GetChildChildrenList(ltctn_list)

...
```

See also

BuildModel
GetColumn

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 )`

<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 find the properties for a column.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>An integer whose value is the index of the column for which you want to find properties.</td>
</tr>
<tr>
<td><code>label</code></td>
<td>A string identifying the label of the column for which you want to find properties. This argument is passed by reference.</td>
</tr>
</tbody>
</table>
| `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

IF lv_list.View <> ListViewReport! THEN
  lv_list.View = ListViewReport!
END
```
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

GetCommandDDE

Description
Obtains the command sent by the client application when your application is a DDE server.

Syntax
GetCommandDDE (string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string variable in which GetCommandDDE will store the command</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, GetCommandDDE returns null.

Usage
When a DDE client application sends a command to your application, the action triggers a RemoteExec event in the active window. In that event’s script, you call GetCommandDDE to find out what command has been sent. You decide how your application will respond to the command.

To enable DDE server mode, use the function StartServerDDE, in which you decide how your application will be known to other applications.

Examples
This excerpt from a script for the RemoteExec event checks to see if the action requested by the DDE client is Open Next Sheet. If it is, the DDE server opens another instance of the sheet DataSheet. If the requested action is Shut Down, the DDE server shuts itself down. Otherwise, it lets the DDE client know the requested action was invalid.
The variables `ii_sheetnum` and `i_DataSheet[ ]` are instance variables for the window that responds to the DDE event:

```power_script
integer ii_sheetnum
DataStream i_DataSheet[ ]
```

This script that follows uses the local variable `ls_Action` to store the command sent by the client application:

```power_script
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**

```power_script
GetCommandDDEOrigin( applstring )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>applstring</code></td>
<td>A string variable in which <code>GetCommandDDEOrigin</code> 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.

**Usage**

The server application calling this function can use the application name (its own DDEname) to determine if it wants to respond to this command. Otherwise, the function provides no additional information about the client.
GetCompanyName

Examples

This script uses the local variable `ls_name` to store the name the client application used to identify the server application:

```sql
string ls_name
GetCommandDDEOrigin(ls_name)
```

See also

GetCommandDDE
StartServerDDE
StopServerDDE

GetCompanyName

Description

Returns the company name for the current execution context.

Applies to

ContextInformation objects

Syntax

```sql
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:

```sql
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
    s1e_co_name.text = ls_company
END IF
```

See also

GetContextService
GetFixesVersion
GetHostObject
GetMajorVersion
GetMinorVersion
GetName
GetShortName
GetVersionName

GetContextKeywords

Description
Retrieves one or more values associated with a specified keyword.

Applies to
ContextKeyword objects

Syntax
\texttt{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 \texttt{values} if the function succeeds and -1 if an error occurs.

Usage
Call this function to access environment variables. Environment-variable availability differs by execution context:

- **PowerBuilder runtime**  The function accesses DOS environment variables, each of which has a unique keyword.

- **PowerBuilder window plug-in**  The function accesses keywords specified in the EMBED tag. These keywords need not be unique. If there is no keyword specified in the EMBED tag, the function attempts to access a DOS environment variable with the specified name.

- **PowerBuilder window ActiveX**  The function accesses DOS environment variables, each of which has a unique keyword.

Examples
This example calls the GetContextKeywords function:

```powerbuilder
String ls_keyword
 Integer li_count, li_return
 ContextKeyword lcx_key

 li_return = this.GetContextService &
 (*"ContextKeyword", lcx_key)
 ls_keyword = sle_name.Text
```
GetContextService

1cx_key.GetContextKeywords &
(ls_keyword, is_values)
FOR li_count = 1 to UpperBound(is_values)
    lb_parms.AddItem(is_values[li_count])
NEXT
See also GetContextService

GetContextService

Description Creates a reference to a context-specific instance of the specified service.
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 – Context keyword service</td>
</tr>
<tr>
<td></td>
<td>• CORBACurrent – CORBA current service for client- or</td>
</tr>
<tr>
<td></td>
<td>component-management of EAServer transactions</td>
</tr>
<tr>
<td></td>
<td>• ErrorLogging – Error logging service for PowerBuilder</td>
</tr>
<tr>
<td></td>
<td>components running in a transaction server such as EAServer</td>
</tr>
<tr>
<td></td>
<td>or COM+</td>
</tr>
<tr>
<td></td>
<td>• Internet – Internet service</td>
</tr>
<tr>
<td></td>
<td>• SSLServiceProvider – SSL service provider service that</td>
</tr>
<tr>
<td></td>
<td>allows PowerBuilder clients to establish SSL connections to</td>
</tr>
<tr>
<td></td>
<td>EAServer components</td>
</tr>
<tr>
<td></td>
<td>• TransactionServer – Transaction server service for</td>
</tr>
<tr>
<td></td>
<td>PowerBuilder components running in a transaction server</td>
</tr>
<tr>
<td></td>
<td>such as EAServer or COM+</td>
</tr>
<tr>
<td>servicereference</td>
<td>PowerObject into which the function places a reference to the</td>
</tr>
<tr>
<td></td>
<td>service object specified by servicename. This argument is</td>
</tr>
<tr>
<td></td>
<td>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.
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:

```powerscript
Integer li_return
ContextKeyword lcx_key

li_return = this.GetContextService &
    ("ContextKeyword", 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
sslcallback.GetCredentialAttribute ( thesessioninfo, attr, attrvalues )

<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_INFILE</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>

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 useEntertrustId 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.

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:

```powerscript
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
```
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:

```powerbuilder
sle_1.text = GetCurrentDirectory ( )
```

See also
ConnectToServer
GetCertificateLabel
GetPin
TrustVerify

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 )
```
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`:

```power_script
integer SeriesNbr
double data_value

// Get the number of the series.
SeriesNbr = &
dw_equipment.FindSeries(“gr_computers”, “Costs”)
data_value = dw_equipment.GetData( &
”gr_computers”, SeriesNbr, 3)
```

These statements obtain the data value of the data point under the mouse pointer in the graph `gr_prod_data` and store it in `data_value`:

```power_script
integer SeriesNbr, ItemNbr
double data_value
grObjectType MouseHit

MouseHit = &
gr_prod_data.ObjectAtPointer(SeriesNbr, ItemNbr)
IF MouseHit = TypeSeries! THEN
    data_value = &
gr_prod_data.GetData(SeriesNbr, ItemNbr)
END IF
```
These statements obtain the x value of the data point in the scatter graph `gr_sales_yr` and store it in `data_value`:

```pascal
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**

Gets the unformatted text from an EditMask control.

**Description**

Applies to EditMask controls

**Syntax**

```
editmaskname.GetData ( datavariablename )
```

<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>datavariablename</code></td>
<td>A variable to which <code>GetData</code> will assign the unformatted data in the EditMask control. The datatype of <code>datavariablename</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 datattype 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`:

```pascal
date d
em_date.GetData(d)
sle_date.Text = String(d, "mm-dd-yy")
```
This example gets string data from the EditMask control `em_string` and assigns the result to `sle_string`. Characters in the edit mask are ignored:

```powerscript
string s
em_string.GetData(s)
sle_string.Text = s
```

### Syntax 3

#### For data in an OLE server

**Description**

Gets data from the OLE server associated with an OLE control using Uniform Data Transfer.

**Applies to**

OLE controls and OLE custom controls

**Syntax**

```
olename.GetData(clipboardformat, data)
```

<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>
<tr>
<td><code>data</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
**GetData**

| 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`: |
| | ```
| string ls_oledata
| integer li_rtn
| li_rtn = ole_word6.GetData( &
| | ClipFormatText!, ls_oledata)
| mle_text.Text = ls_oledata
| ``` |
| | One OLE control displays a Microsoft Word document containing a table of data. This example gets the data in the report and assigns it to a graph in a second OLE control. Microsoft Graph in the second control interprets the first row in the table as headings, and subsequent rows as categories or series, depending on the settings on the Data menu: |
| | ```
| string ls_data
| integer li_rtn
| li_rtn = ole_word.GetData(ClipFormatText!, ls_data)
| IF li_rtn <> 1 THEN RETURN
| li_rtn = ole_graph.SetData(ClipFormatText!, ls_data)
| ``` |
| See also | SetData |
## GetDataDDE

**Description**
Obtains data sent from another DDE application and stores it in the specified string variable. PowerBuilder can use `GetDataDDE` when acting as a DDE client or a DDE server application.

**Syntax**
```
GetDataDDE ( string )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
<td>A string variable in which <code>GetDataDDE</code> 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.

**Syntax**
```
GetDataDDEOrigin ( applstring, topicstring, itemstring )
```
GetDataDDEOrigin

**Argument** | **Description**
---|---
**applstring** | A string variable in which `GetDataDDEOrigin` will store the name of the server application.
**topicstring** | A string variable in which `GetDataDDEOrigin` will store the topic (for example, in Microsoft Excel, the topic could be `REGION.XLS`).
**itemstring** | A string variable in which `GetDataDDEOrigin` will store the item identification (for example, in Microsoft Excel, the item could be `R1C2`).

**Return value**

Integer. Returns 1 if it succeeds and -1 if an error occurs (such as the function was called in the wrong context). If any argument’s value is null, `GetDataDDEOrigin` returns null.

**Usage**

Call `GetDataDDEOrigin` in the window-level script for a RemoteSend event or a HotLinkAlarm event.

When your application is a DDE server, call `GetDataDDEOrigin` in the script for the RemoteSend event. Use it to determine the topic and item requested by the client. The application name is the application specified by the client (the server’s own DDEname).

When your application is a DDE client, call `GetDataDDEOrigin` in the script for the HotLinkAlarm event. Use it to identify the source of the data when hot links may exist for more than one topic within the server application or for more than one application.

**Examples**

This example illustrates how to call `GetDataDDEOrigin`:

```powershell
string WhichAppl, WhatTopic, WhatLoc
GetDataDDEOrigin(WhichAppl, WhatTopic, WhatLoc)
```

**See also**

`GetDataDDE`
`OpenChannel`
`StartServerDDE`
`StopServerDDE`
**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 )
```

**Argument** | **Description**
---|---
controlname | The name of the graph for which you want the percentage a pie slice is exploded, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only) | (Optional) A string whose value is the name of the graph in the DataWindow control for which you want the percentage a pie slice is exploded
series | The number that identifies the series
datapoint | The number of the exploded data point (that is, the pie slice)
percentage | An integer variable in which you want to store the percentage of the graph’s radius that the pie slice is exploded

**Return value**
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, GetDataPieExplode returns null.

**Examples**
This example reports the percentage that a pie slice is exploded when the user clicks on that slice. The code checks whether the graph is a pie graph using the property Graphite. It then finds out whether the user clicked on a pie slice by checking the series and data point values set by ObjectAtPointer. The script is for the DoubleClicked event of a graph object:

```
integer series, datapoint
gObjectclickedtype
type 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("Expllosion 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**
--- | ---
controlname | The name of the graph for which you want the color of a data point, or the name of the DataWindow control containing the graph.
graphcontrol | (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)
seriesnumber | The number of the series in which you want the color of a data point.
datapointnumber | The number of the data point for which you want the color.
colortype | A value of the `grColorType` enumerated datatype specifying the aspect of the data point for which you want the color. Values are:
- Background! – The background color
- Foreground! – Text (fill color)
- LineColor! – The color of the line
- Shade! – The shaded area of three-dimensional graphics

**colorvariable** | A long variable in which you want to store the color.

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. Stores a color value in `colorvariable`. If any argument’s value is null, `GetDataStyle` returns null.

Examples
This example gets the text (foreground) color used for data point 6 in the series named Salary in the graph `gr_emp_data`. It stores the color value in the variable `color_nbr`:

```
long color_nbr
integer SeriesNbr

// Get the number of the series
SeriesNbr = gr_emp_data.FindSeries("Salary")

// Get the color
gr_emp_data.GetDataStyle(SeriesNbr, 6, & Foreground!, color_nbr)
```
**GetDataStyle**

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`:

```plaintext
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.

**Description**

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><code>controlname</code></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><code>graphcontrol</code> (DataWindow control only)</td>
<td>(Optional) A string whose value is the name of the graph (in the DataWindow control) for which you want the line style and width of a data point.</td>
</tr>
<tr>
<td><code>seriesnumber</code></td>
<td>The number of the series in which you want 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 the line style and width.</td>
</tr>
<tr>
<td><code>linestyle</code></td>
<td>A variable of type LineStyle in which you want to store the line style.</td>
</tr>
<tr>
<td><code> linewidth</code></td>
<td>An integer variable in which you want to store the width of the line. The width is measured in pixels.</td>
</tr>
</tbody>
</table>
Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. For the specified series and data point, stores its line style in linestyle and the line’s width in linewidth. If any argument’s value is null, GetDataStyle returns null.

Usage

For the enumerated datatype values that GetDataStyle will store in linestyle, see SetDataStyle.

Examples

This example gets the line style and width of data point 10 in the series named Costs in the graph gr_product_data. It stores the information in the variables linestyle and linewidth:

```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 sle_series in the graph gr_depts in the DataWindow control dw_employees. The information is stored in the variables linestyle and linewidth:

```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

FindSeries
GetSeriesStyle
SetDataStyle
SetSeriesStyle
GetDataStyle

Syntax 3

**For the fill pattern or symbol of a data point**

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 )
```

**Argument**

<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)
```
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`:

```powerlanguage
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`:

```powerlanguage
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`:

```powerlanguage
integer SeriesNbr
grSymbolType symbol_type

// Get the number of the series
SeriesNbr = dw_employees.FindSeries("gr_depts", &
  sle_series.Text)

// Get the symbol
dw_employees.GetDataStyle("gr_depts", SeriesNbr, &
  6, symbol_type)
```

See also

- `FindSeries`
- `GetSeriesStyle`
- `SetDataStyle`
- `SetSeriesStyle`
GetDataValue

Description
Obtains the value of a data point in a series in a graph.

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

Syntax
`controlname.GetDataValue ( { graphcontrol, } seriesnumber, datapoint, datavariable {, xory } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></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><code>graphcontrol</code></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><code>seriesnumber</code></td>
<td>The number that identifies the series from which you want data.</td>
</tr>
<tr>
<td><code>datapoint</code></td>
<td>The number of the data point for which you want the value.</td>
</tr>
<tr>
<td><code>datavariable</code></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>
</tbody>
</table>
| `xory`         | (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) |

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, GetDataValue returns null.

Usage
GetDataValue retrieves data from any graph. The data is stored in `datavariable`, whose datatype must match the datatype of the graph’s values axis. If the values axis is numeric, you can also use the `GetDataValue` 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`:

```powershell
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)
```

516
These statements obtain the data value of the data point under the mouse pointer in the graph \texttt{gr\_prod\_data} and store it in \texttt{data\_value}. If the user does not click on a data point, then \texttt{ItemNbr} is set to 0. The categories of the graph are time values:

\begin{verbatim}
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
\end{verbatim}

These statements obtain the x value of the data point in the scatter graph \texttt{gr\_sales\_yr} and store it in \texttt{data\_value}. If the user does not click on a data point, then \texttt{ItemNbr} is set to 0. The datatype of the category axis is Date:

\begin{verbatim}
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
\end{verbatim}

See also
\begin{itemize}
  \item DeleteData
  \item FindSeries
  \item InsertData
  \item ObjectAtPointer
\end{itemize}
GetDateLimits

Description
Retrieves the maximum and minimum date limits specified for the calendar.

Applies to
MonthCalendar control

Syntax
controlname.GetDateLimits( min, max )

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:

```
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
    str1 = "Minimum date is " + string(mindate)
    str2 = "Maximum date is " + string(maxdate)
end if

MessageBox("Date Limits", str1 + "\n\n" + str2)
```

See also
SetDateLimits
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 } )`

Argument | Description
---|---
controlname | The name of the MonthCalendar control for which you want the range of dates
start | A date specifying the first date in the displayed range returned by reference
end | A date specifying the last date in the displayed range returned by reference
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.
Examples

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:

```
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:

```
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**

\[ \text{DynamicDescriptionArea}.\text{GetDynamicDate}(\ index ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{DynamicDescriptionArea}</td>
<td>The name of the DynamicDescriptionArea, usually SQLDA.</td>
</tr>
<tr>
<td>\textit{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 \textit{index} in \textit{DynamicDescriptionArea}. Returns 1900-01-01 if an error occurs. If any argument’s value is null, \textit{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 \textit{GetDynamicDate} when the value of OutParmType is TypeDate! for the value in the array that you want to retrieve.

**Examples**

These statements set Today to the Date data in the second output parameter descriptor:

```
Date Today
Today = GetDynamicDate(SQLDA, 2)
```

If you have executed Format 4 dynamic SQL statements, data is stored in the DynamicDescriptionArea. This example finds out the datatype of the stored data and uses a \texttt{CHOOSE CASE} statement to assign it to local variables.

If the \texttt{SELECT} statement is:

```
SELECT emp_start_date FROM employee;
```

then the code at \texttt{CASE TypeDate!} will be executed.

For each case, other processing could assign the value to a DataWindow so that the value would not be overwritten when another value has the same ParmType:

```
Date Datevar
Time Timevar
DateTime Datetimevar
Double Doublevar
String Stringvar
```
FOR n = 1 to SQLDA.NumOutputs
    CHOOSE CASE SQLDA.OutParmType[n]
    CASE TypeString!
        Stringvar = SQLDA.GetDynamicString(n)
        ... // Other processing
    CASE TypeDecimal!, TypeDouble!, &
        TypeInteger!, TypeLong!, &
        TypeReal!, TypeBoolean!
        Doublevar = SQLDA.GetDynamicNumber(n)
        ... // Other processing
    CASE TypeDate!
        Datevar = SQLDA.GetDynamicDate(n)
        ... // Other processing
    CASE TypeDateTime!
        Datetimevar = SQLDA.GetDynamicDateTime(n)
        ... // Other processing
    CASE TypeTime!
        Timevar = SQLDA.GetDynamicTime(n)
        ... // Other processing
    CASE ELSE
        MessageBox("Dynamic SQL", &
            "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.
CHAPTER 10 PowerScript Functions

DynamicDescriptionArea.GetDynamicDateTime (index)

Syntax

**DynamicDescriptionArea.GetDynamicDateTime (index)**

**Argument** | **Description**
--- | ---
DynamicDescriptionArea | The name of the DynamicDescriptionArea, usually SQLDA.
index | An integer identifying the output parameter descriptor from which you want to get the data. *Index* must be less than or equal to the value in NumOutputs in DynamicDescriptionArea.

Return value

DateTime. Returns the DateTime data in the output parameter descriptor identified by *index* in DynamicDescriptionArea. Returns 1900-01-01 00:00:00.000000 if an error occurs. If any argument’s value is null, GetDynamicDateTime returns null.

Usage

Use GetDynamicDateTime when the value of OutParmType is TypeDateTime! for the value that you want to retrieve from the array.

To test for the error value, you must use the DateTime function to construct the value to which you want to compare the returned value. PowerBuilder does not support DateTime literals.

Examples

These statements set *SystemDateTime* to the DateTime data in the second output parameter descriptor:

```powerbuilder
DateTime SystemDateTime
SystemDateTime = SQLDA.GetDynamicDateTime(2)
IF SystemDateTime = &DateTime(1900-01-01, 00:00:00) THEN
    ... // Error handling
END IF
```

For an example of retrieving data from the DynamicDescriptionArea, see GetDynamicDate.

See also

GetDynamicDate
GetDynamicNumber
GetDynamicString
GetDynamicTime
SetDynamicParm
Using dynamic SQL
GetDynamicNumber

**Description**
Obtains numeric data from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

**Restriction**
You can use this function *only* after executing Format 4 dynamic SQL statements.

**Syntax**

\[
\text{DynamicDescriptionArea}.\text{GetDynamicNumber} \left( \text{index} \right)
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{DynamicDescriptionArea})</td>
<td>The name of the DynamicDescriptionArea, usually SQLDA.</td>
</tr>
<tr>
<td>(\text{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**
Double. Returns the numeric data in the output parameter descriptor identified by \(\text{index}\) in \(\text{DynamicDescriptionArea}\). Returns 0 if an error occurs. If any argument's value is null, GetDynamicNumber returns null.

**Usage**
Use GetDynamicNumber when the value of OutParmType is TypeInteger!, TypeDecimal!, TypeDouble!, TypeLong!, TypeReal!, or TypeBoolean! for the value that you want to retrieve from the array.

If you use this function to return a decimal value, expect the value to be rounded when you convert the double datatype back to a decimal. Doubles do not support the same precision as decimals.

**Examples**
These statements set \(\text{DeptId}\) to the numeric data in the second output parameter descriptor:

\[
\begin{align*}
\text{Integer DeptId} \\
\text{DeptId} &= \text{SQLDA}.\text{GetDynamicNumber}(2)
\end{align*}
\]

For an example of retrieving data from the DynamicDescriptionArea, see GetDynamicDate.

**See also**
GetDynamicDate  
GetDynamicDateTime  
GetDynamicString  
GetDynamicTime  
SetDynamicParm  
Using dynamic SQL
CHAPTER 10  PowerScript Functions

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
\[ \text{DynamicDescriptionArea}.\text{GetDynamicString} \left( \text{index} \right) \]

<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
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:

\[
\begin{align*}
\text{String LName} \\
\text{LName} &= \text{SQLDA.GetDynamicString}(2)
\end{align*}
\]

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:

```
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)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>environmentinfo</td>
<td>The name of the Environment object that will hold the information about the environment</td>
</tr>
</tbody>
</table>

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:

```powershell
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)
```

---

**Get*FileOpenName***

**Description**  Displays the system’s Open File dialog box and allows the user to select a file or enter a file name.

**Syntax**

```powershell
GetFileOpenName ( title, pathname, filename {, extension {, filter {, initdir {, aFlag } } } } )
GetFileOpenName ( title, pathname, filename[ ], extension {, filter {, initdir {, aFlag } } } )
```

**Argument Description**

<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>
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 $2^{(\text{index} - 1)}$, 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 <code>GetOpenFileName</code> 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.

```powerscript
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," &
```
GetFileOpenName

    + "All Files (*.*)", *.txt", &
    "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
CHAPTER 10  PowerScript Functions

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

GetFileSaveName ( title, pathname, filename \{, extension \{, filter \{ , initdir \{ , aFlag \} \} \} \} )

GetFileSaveName ( title, pathname, filename \{, extension \{, filter \{ , initdir \{ , aFlag \} \} \} \} )

Argument Description

<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>.</em> The default is: &quot;All Files (<em>.</em>)&quot;,<em>.</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 GetOpenFileName is the sum of the individual option flags. See the table in the Usage section for GetOpenFileName 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.
**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:

```plaintext
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
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 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:

```powerbuilder
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

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:

```
GraphicObject which_control
SingleLineEdit sle_which
CommandButton cb_which
string text_value

which_control = GetFocus()

CHOOSE CASE TypeOf(which_control)

CASE CommandButton!
  cb_which = which_control
  text_value = cb_which.Text

CASE SingleLineEdit!
  sle_which = which_control
  text_value = sle_which.Text

CASE ELSE
  text_value = ""
END CHOOSE
```

**See also**

IsValid
SetFocus
GetFolder

Description
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:

```powerbuilder
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

Description
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)

Argument | Description
--- | ---
sslserviceprovider | Reference to the SSLServiceProvider service instance.
property | The name of the SSL property for which you want to return values.
For a complete list of supported SSL properties, see your EAServer documentation or the online Help for the Connection object.
values | An array of string values for the specified SSL property.

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:

```powerscript
SSLServiceProvider ssl
string ls_values[]
long rc
...
this.GetContextService("SSLServiceProvider", ssl)
rc = ssl.GetGlobalProperty("sessLingerTime", ls_values)
...
```

See also
SetGlobalProperty
**GetHostObject**

**Description**
Provides a reference to the context’s host object.

**Host object support**
Currently, host object support is implemented only in the window ActiveX when running under Internet Explorer. In this situation `GetHostObject` returns a reference to the IWebBrowserApp ActiveX automation server object.

**Applies to**
ContextInformation objects

**Syntax**

```powerbuilder
servicereference.GetHostObject ( hostobject )
```

**Argument** | **Description**
--- | ---
`servicereference` | Reference to the Context Information service instance
`hostobject` | PowerObject into which the function places a reference to the ActiveX automation server object

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

**Usage**
Call this function to obtain a reference to the context object model. If running the window ActiveX under Internet Explorer 3.0 or greater and `hostobject` is an uninstantiated OleObject variable, the function returns a reference to an ActiveX automation server object, which you can use to control the hosting browser. If host object support is not available, the function returns -1 and `hostobject` is null.

**Examples**
This example calls the `GetHostObject` function. `ici_info` is an instance variable of type ContextInformation, which has been populated using the GetContextService function; `ole1` is an instance variable of type OLEObject:

```powerbuilder
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.

### Syntax 1 For ListView controls

**Description**
Retrieves a ListViewItem object from a ListView control so you can examine its properties.

**Applies to**
ListView controls

**Syntax**
```
listviewname.GetItem(index, {column}, item)
```

<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 ListView 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.
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:

```powerbuilder
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

Retrieves the value displayed for a ListView item in a specified column.

**Applies to**

ListView controls

**Syntax**

```
listviewname.GetItem( index, column, label )
```

**Argument** | **Description**
--- | ---
`listviewname` | The name of the ListView control from which you want to retrieve a displayed value.
`index` | The index number of the item for which you want to retrieve a displayed value.
`column` | 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.
`label` | A string variable in which you store the displayed value.

**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.

```powerbuilder
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
CHAPTER 10  PowerScript Functions

Syntax 3

**Description**
Retrieves the data associated with the specified item.

**Applies to**
TreeView controls

**Syntax**
```
treeviewname.GetItem (itemhandle, item)
```

**Argument** | **Description**
--- | ---
`treeviewname` | The name of the TreeView control in which you want to get data for a specified item
`itemhandle` | The handle for the item for which you want to retrieve information
`item` | A TreeViewItem variable in which you want to store the item identified by the item handle

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

**Usage**
Use `GetItem` to retrieve the state information associated with a specific 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:

```
  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:

```
  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()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></td>
<td>The name of the control whose handle or index you want to obtain.</td>
</tr>
</tbody>
</table>

**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:
```
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 InvokePBFunction or TriggerPBEvet function.

Applies to
Window ActiveX controls

Syntax
activexcontrol.GetLastReturn()

Argument | Description
--- | ---
activexcontrol | Identifier for the instance of the PowerBuilder window ActiveX control. When used in HTML, the ActiveX control is the NAME attribute of the object element. When used in other environments, this references the control that contains the PowerBuilder window ActiveX.

Return value
Any. Returns the last return value.

Usage
Call this function after calling InvokePBFunction or TriggerPBEvet to access the return value. JavaScript scripts must use this function to access return values from InvokePBFunction and TriggerPBEvet. VBScript scripts can either use this function or access the return value using an argument in InvokePBFunction or TriggerPBEvet.

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:

```vbnet
... 
  retcd = PBRX1.TriggerPBEvent(theEvent, & numargs, args)
  rc = PBRX1.GetLastReturn()
  IF rc <> 1 THEN
    msgbox "Error. Empty string."
  END IF

  ...
```

See also
GetArgElement
InvokePBFunction
SetArgElement
TriggerPBEvet
## 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:

```
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)`

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
    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
GetMinorVersion
GetName
GetShortName
GetVersionName
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:

```
    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, wf_acos. The user-defined function catches a runtime error—preventing the application from terminating—and then sets the message for a user object, uo_exception, that inherits from the Exception object type:

```
    TRY
       wf_acos()
       CATCH (uo_exception u_ex)
           MessageBox("Out of Range", u_ex.GetMessage())
    END TRY
```

Code for the wf_acos function is shown in the SetMessage function.

See also
SetMessage
## GetMinorVersion

**Description**
Returns the minor version for the current PowerBuilder execution context. For example, at maintenance level 10.2.1 the minor version is 2.

**Applies to**
ContextInformation objects

**Syntax**
```
servicereference.GetMinorVersion( minorversion )
```

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

**Usage**
Call this function to determine the current minor version.

**Examples**
This example calls the `GetMinorVersion` function:
```
String ls_name
Constant String ls_currver = "8.0.3"
Integer li_majver, li_minver, li_fixver
ContextInformation ci

this.GetContextService("ContextInformation", ci)

ci.GetMajorVersion(li_majver)

\[\text{ci.GetMinorVersion}(\text{li_minver})\]

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

<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>
**GetName**

**Description**

Gets the name for the current execution context.

**Applies to**

ContextInformation objects

**Syntax**

```
servicereference.GetName(name)
```

**Argument** | **Description**
---|---
`servicereference` | Reference to the ContextInformation service instance.
`name` | String into which the function places the name. This argument is passed by reference.

**Return value**

Integer. Returns 1 if the function succeeds and -1 if an error occurs. The function returns values as follows:

- **PowerBuilder runtime**: PowerBuilder Runtime
- **PowerBuilder window plug-in**: PowerBuilder window Plug-in
- **PowerBuilder window ActiveX**: PowerBuilder Runtime ActiveX

**Usage**

Call this function to determine the current execution environment.

**Examples**

This example calls the GetName function. `ci` is an instance variable of type ContextInformation:

```powershell
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

\[
\text{olename} \cdot \text{GetNativePointer} ( \text{pointer} )
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{olename}</td>
<td>The name of the OLE control containing the object for which you want the native pointer.</td>
</tr>
<tr>
<td>\text{pointer}</td>
<td>A UnsignedLong variable in which you want to store the pointer. If GetNativePointer cannot get a valid pointer, \text{pointer} is set to 0.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage
\( \text{Pointer} \) 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.

Examples
This example gets a pointer for the OLECustomControl ocx_spell for making external function calls for OLE automation:

```
UnsignedLong lul_oleptr
integer li_rtn

li_rtn = ocx_spell.GetNativePointer(lul_oleptr)
IF li_rtn = 0 THEN
    ... // Call external functions for automation
    ocx_spell.ReleaseNativePointer(lul_oleptr)
END IF
```

See also
GetAutomationNativePointer
ReleaseAutomationNativePointer
ReleaseNativePointer
**GetNextSheet**

**Description**
Obtains the sheet that is behind the specified sheet in the MDI frame.

**Applies to**
MDI frame windows

**Syntax**
```
mdiframewindow.GetNextSheet ( sheet )
```

**Argument** | **Description**
--- | ---
`mdiframewindow` | The MDI frame window in which you want the next sheet
`sheet` | The sheet for which you want the sheet that is behind it

**Return value**
Window. Returns the sheet that is behind `sheet` in the MDI frame. If there is no sheet behind `sheet`, `GetNextSheet` returns an invalid value. If any argument’s value is `null`, `GetNextSheet` returns `null`.

**Usage**
To cycle through the open sheets in a frame, use `GetFirstSheet` to get the front sheet and `GetNextSheet` one or more times to get the rest of the sheets. Test each return value with `IsValid` to see if you have reached the last sheet. Do not use `GetFirstSheet` and `GetNextSheet` in combination with `GetActiveSheet`.

---

**Did GetNextSheet return a valid window?**

Use the `IsValid` function to find out if `GetNextSheet` returned a valid window. If there is no sheet behind the one you specified, the return value is not valid.

---

**Examples**
The following script for a menu selection loops through the open sheets in front-to-back order and displays the names of the open sheets in the ListBox `lb_sheets`:

```powerbuilder
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
END IF
```

**See also**
- `GetFirstSheet`
- `IsValid`
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)`

<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 find the coordinates of the upper-left corner</td>
</tr>
<tr>
<td>x</td>
<td>An integer variable in which you want to store the X coordinate for the ListView control</td>
</tr>
<tr>
<td>y</td>
<td>An integer variable in which you want to store the Y coordinate for the ListView control</td>
</tr>
</tbody>
</table>

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:

```powerscript
integer li_index
listviewitem l_lvi

li_index = lv_list.SelectedIndex()
lv_list.GetItem(li_index, l_lvi)

lv_list.GetOrigin(l_lvi.ItemX, l_lvi.ItemY)

sle_info.Text = "X is " + String(l_lvi.ItemX) & 
                + " and Y is " + String(l_lvi.ItemY)

st_clock.Move(l_lvi/itemx, l_lvi.ItemY)

MessageBox("Clock Location", "X is " & 
            + String(st_clock.X) & 
            + ", and Y is " & 
            + String(st_clock.Y)+")."
```
GetParagraphSetting

Description
Gets the size of the indentation, left margin, or right margin of the paragraph containing the insertion point in a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
rtecontrol.GetParagraphSetting (whichsetting)

Argument | Description
--- | ---
rtecontrol | The name of the control for which you want paragraph information.
whichsetting | A value of the ParagraphSetting enumerated 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:

```powerbuilder
long ll_indent
ll_indent = rte_1.GetParagraphSetting(Indent!)
```

See also
GetAlignment
GetSpacing
GetTextColor
GetTextStyle
SetParagraphSetting

GetParent

Description
Obtains the parent of the specified object.

Applies to
Any object

Syntax
objectname.GetParent ( )

Argument | Description
--- | ---
objectname | A control in a window or user object or an item on a menu for which you want the parent object

Return value
PowerObject. Returns a reference to the parent of objectname.
Examples

In event scripts for a user object that will be used as a tab page, you can use code like the following to make references to the parent Tab control generic:

```powerscript
// 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:

```powerscript
// 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 CTSecurity::UserAbortedException to the server. You need to catch the exception by wrapping the ConnectToServer function in a try-catch block.
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:

```power_script
//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)

// otherwise, abort..
else
    ue_cancelled = CREATE userabortedexception
    ue_cancelled.text = "User cancelled request when " &
        + "asked for PIN."
    throw ue_cancelled
end if
return s_PIN
```

See also
ConnectToServer
GetCertificateLabel
GetCredentialAttribute
TrustVerify
GetRecordSet

Returns the current ADO Recordset object.

Applies to
ADOResultSet objects

Syntax
adoresultset.GetRecordSet ( adorecordsetobject )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adoresultset</td>
<td>An ADOResultSet object that contains an ADO Recordset.</td>
</tr>
<tr>
<td>adorecordsetobject</td>
<td>An OLEObject object into which the function places the current ADO Recordset. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

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.

```pascal
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
loo_ADOrecordset = CREATE OLEObject
loo_ADOrecordset.GetRecordSet(lrs_ADOrecordset)
// Call native ADO Recordset methods on the OLEObject
loo_ADOrecordset.MoveFirst()
```

See also
GenerateResultSet method for DataWindows in the DataWindow Reference or the online Help
SetRecordSet
SetResultSet
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.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make a single request of a DDE server application (called a cold link)</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Request data from a DDE server application after you have opened a channel (called a warm link)</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

### Syntax 1

**For single DDE requests**

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**

```powerscript
GetRemote(location, target, applname, topicname)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>location</code></td>
<td>A string whose value is the location of the data you want returned from the DDE server application. The format of <code>location</code> depends on the particular DDE server application that will receive the message.</td>
</tr>
<tr>
<td><code>target</code></td>
<td>A string variable into which the returned data will be placed.</td>
</tr>
<tr>
<td><code>applname</code></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 <code>StartServerDDE</code> function call.</td>
</tr>
<tr>
<td><code>topicname</code></td>
<td>A string identifying the data or the instance of the application you want to use with the command (for example, in Microsoft Excel, the topic name could be system or the name of an open spreadsheet). If another PowerBuilder application is the DDE server, this is the topic specified in its <code>StartServerDDE</code> function call.</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

If any argument’s value is null, GetRemote returns null.
GetRemote

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.

Syntax
```
GetRemote ( location, target, handle {{, windowhandle} })
```

<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. The format of the location depends on the DDE application that will receive the request.</td>
</tr>
<tr>
<td>target</td>
<td>A PowerBuilder string variable into which the returned data will be placed.</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</td>
<td>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.</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 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:

```powerscript
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.

**Examples**  
This example retrieves the selected date into `seldate`:

```powerbuilder
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>controlname</td>
<td>The name of the MonthCalendar control for which you want to determine the range of selected dates</td>
</tr>
<tr>
<td>start</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>end</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:

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
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.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the series’ colors</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Get the line style and width used by the series</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Get the fill pattern or symbol for the series</td>
<td>Syntax 3</td>
</tr>
<tr>
<td>Find out if the series is an overlay (a series shown as a line on top of another graph type)</td>
<td>Syntax 4</td>
</tr>
</tbody>
</table>

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**

Obtains the colors associated with a series in a graph.

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

Syntax

```
colorname.GetSeriesStyle( { graphcontrol, } seriesname, colortype, colorvariable )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colorname</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>graphcontrol</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>seriesname</td>
<td>A string whose value is the name of the series for which you want the color.</td>
</tr>
</tbody>
</table>
### GetSeriesStyle

**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:

```powerScript
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:

```powerScript
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:

```powerScript
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)
```

### Argument Description

<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:

```powerScript
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:

```powerScript
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:

```powerScript
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)
```
**GetSeriesStyle**

```powerbuilder
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><code>controlname</code></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><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 line style information.</td>
</tr>
<tr>
<td><code>seriesname</code></td>
<td>A string whose value is the name of the series for which you want the line style information.</td>
</tr>
<tr>
<td><code>linestyle</code></td>
<td>A variable of type LineStyle in which you want to store the line style of <code>seriesname</code>.</td>
</tr>
<tr>
<td><code>linewidth</code></td>
<td>An integer variable in which you want to store the line width for <code>seriesname</code>. 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`:

```powerbuilder
string SeriesName
integer SeriesNbr, Data_Point, line_width
LineStyle line_style
```
grObjectType MouseHit

MouseHit = ObjectAtPointer(SeriesNbr, Data_Point)

IF MouseHit = TypeSeries! THEN
   SeriesName = &
   gr_product_data.SeriesName(SeriesNbr)

   gr_product_data.GetSeriesStyle(SeriesName, &
       line_style, line_width)
END IF

See also
AddSeries
GetDataStyle
FindSeries
SetDataStyle
SetSeriesStyle

Syntax 3
For the fill pattern or symbol of a series

Description
Obtains the fill pattern or symbol of a series in a graph.

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

Syntax
controlname.GetSeriesStyle ( { graphcontrol, } seriesname, enumvariable )

Argument | Description
--- | ---
controlname | The name of the graph for which you want the style information for a series in a graph, or the name of the DataWindow control containing the graph.

graphcontrol (DataWindow control only) | (Optional) A string whose value is the name of the graph in the DataWindow control for which you want the style information.

seriesname | A string whose value is the name of the series for which you want the style information.

enumvariable | The variable in which you want to store the style information. You can specify a FillPattern or grSymbolType variable. The style information that GetSeriesStyle stores depends on the variable type.

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. Stores in enumvariable a value of the appropriate enumerated datatype for the fill pattern or symbol used for the specified series. If any argument’s value is null, GetSeriesStyle returns null.
GetSeriesStyle

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*:

```powerbuilder
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:

```powerbuilder
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
```
controlname.GetSeriesStyle( { graphcontrol, } seriesname, overlayindicator )
```

<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 overlay status of 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 overlay status.</td>
</tr>
<tr>
<td>seriesname</td>
<td>A string whose value is the name of the series for which you want the overlay status.</td>
</tr>
<tr>
<td>overlayindicator</td>
<td>A boolean variable in which you want to store a value indicating whether the series is an overlay. GetSeriesStyle sets overlayindicator to true if the series is an overlay and false if it is not.</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:
```
boolean is_overlay
gr_emp_data.GetSeriesStyle(sle_series.Text, & is_overlay)
```
GetShortName

**Description**

Gets the short name for the current PowerBuilder execution context.

**Applies to**

ContextInformation objects

**Syntax**

```
 serviceref.GetShortName( shortname )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>serviceref</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. The function returns values as follows:

- **PowerBuilder runtime**  PBRun
- **PowerBuilder window plug-in**  PBWinPlugin
- **PowerBuilder window ActiveX**  PBRTX

**Usage**

Call this function to determine the current execution environment.

**Examples**

This example calls the GetShortName function. `ci` is an instance variable of type ContextInformation:

```
String ls_name

this.GetContextService("ContextInformation", ci)

   ci.GetShortName( ls_name )

   IF ls_name <> "PBRun" THEN
       cb_close.visible = FALSE
   END IF
```

**See also**

GetCompanyName
GetContextService
GetFixesVersion
GetHostObject
GetMajorVersion
GetMinorVersion
GetName
GetVersionName
GetSpacing

Description
Obtains the line spacing of the paragraph containing the insertion point in a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
rtename.GetSpacing()

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
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

Syntax
CORBACurrent.GetStatus()

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
GetStatus

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. EASever 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.

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:

```plaintext
// 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:9000")
IF li_rc <> 1 THEN
   // handle the error
ELSE
   li_status = corbcurr.GetStatus()
```

PowerBuilder
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

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:

    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**
```powerbuilder
rename.GetTextColor()
```

**Argument** | **Description**
---|---
rename | The name of the RichTextEdit control in which you want to find out the color of selected text

**Return value**
Long. Returns the long value that specifies the color of the currently selected text. If text of different colors is selected, GetTextColor returns the color of the first selected character. GetTextColor returns -1 if an error occurs.

**Examples**
This example stores a long representing the color of the selected text in rte_1:

```powerbuilder
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**
```powerbuilder
rename.GetTextStyle( textstyle )
```

**Argument** | **Description**
---|---
rename | The name of the RichTextEdit control in which you want to find the formatting of selected text.

textstyle | A value of the enumerated datatype TextStyle specifying the text style you want to check for. Values are:

- Bold!
- Italic!
- Strikeout!
- Subscript!
- Superscript!
- Underlined!
Return value

Boolean. Returns true if the selected text is formatted with the specified text style and false if it is not. If textstyle is null, GetTextStyle returns null.

Usage

Text can be formatted with more than one text style. To test for different styles, call GetTextStyle more than once.

Examples

A previously defined structure is an instance variable istr_text for the current window. The structure contains the boolean fields: b_isBold, b_isItalic, and b_isUnderlined. This example checks whether the selected text has these styles and stores true or false values in the structure for each style:

istr_text.b_isBold = rte_fancy.GetTextStyle(Bold!)
istr_text.b_isItalic = rte_fancy.GetTextStyle(Italic!)
istr_text.b_isUnderlined = &
rte_fancy.GetTextStyle(Underlined!)

See also

GetTextColor
SetSpacing
SetTextColor
SetTextStyle

GetToday

Description

Returns the value that the calendar uses as today’s date.

Applies to

DatePicker, MonthCalendar controls

Syntax

controlname.GetToday ( )

<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 today’s date</td>
</tr>
</tbody>
</table>

Return value

Date.

Usage

By default, the current system date is set as the Today date. You can use the SetToday function to specify a different date.

Examples

This example retrieves the Today date in a DatePicker control into currentdate:

Date currentdate
currentdate = dp_1.GetToday()

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

window.GetToolbar (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 for which you want the current settings</td>
</tr>
<tr>
<td>visible</td>
<td>A boolean variable in which you want to store a value indicating whether the toolbar is visible</td>
</tr>
<tr>
<td>alignment</td>
<td>A variable of the ToolbarAlignment enumerated datatype in which you want to store the current alignment of the toolbar (optional)</td>
</tr>
<tr>
<td>floatingtitle</td>
<td>A string variable in which you want to store the toolbar title that is displayed when the alignment is Floating! (optional)</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns 1 if it succeeds. GetToolbar returns -1 if there is no toolbar for the index you specify or if an error occurs. If any argument's value is null, returns null.

Usage

To find out the position of the docked or floating toolbar, call GetToolbarPos.

Examples

This example finds out whether toolbar 1 is visible. It also gets the alignment and title of toolbar 1. The values are stored in the variables lb_visible, lta_align, and ls_title:

```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_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_title
             + ls_align + "~r~n" &
### 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**

- **Description**
  Gets the position of a docked toolbar.

- **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><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>dockrow</code></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><code>offset</code></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, <code>offset</code> is measured from the left edge. For toolbars at the left or right, <code>offset</code> is measured from the top.</td>
</tr>
</tbody>
</table>

- **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 `sle_2`. The example gets the toolbar position information and displays it in a MultiLineEdit `mle_1`:

  ```
  integer li_index, li_rtn
  integer li_dockrow, li_offset
  ```
li_index = Integer(sle_2.Text)
li_rtn = w_frame.GetToolbarPos(li_index, &
   li_dockrow, li_offset)

// Report the position settings
IF li_rtn = 1 THEN
   mle_1.Text = String(li_dockrow) + "r-n"
   + String(li_offset)
ELSE
   mle_1.Text = "Can't get toolbar position"
END IF

See also
GetToolbar
SetToolbar
SetToolbarPos

Syntax 2
For floating toolbars
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>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>x</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>y</td>
<td>An integer variable in which you want to store the y coordinate of the floating toolbar. If the toolbar is docked, y is set to the most recent value.</td>
</tr>
<tr>
<td>width</td>
<td>An integer variable in which you want to store the width of the floating toolbar. If the toolbar is docked, width is set to the most recent value.</td>
</tr>
<tr>
<td>height</td>
<td>An integer variable in which you want to store the height of the floating toolbar. If the toolbar is docked, height is set to the most recent value.</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, 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:

```pascal
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()`

<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
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:

```plaintext
// 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 )`

<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 whose source data is returned in <code>data</code></td>
</tr>
<tr>
<td><code>data</code></td>
<td>InternetResult descendant containing an overridden <code>InternetData</code> function that handles the HTML source for <code>urlname</code></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

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:

```powerbuilder
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.

Applies 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:

```powerbuilder
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</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:

```powerbuilder
f function boolean flashwindow(uint hnd, boolean inst) &
   library "user.exe"
```

The code that flashes the window’s title bar is:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
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><code>objectname</code></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

Sets a timer so that PowerBuilder triggers an Application Idle event when there has been no user activity for a specified number of seconds.

Syntax

```
Idle(n)
```

Argument | Description
---|---
$n$ | The number of seconds of user inactivity allowed before PowerBuilder triggers an Application Idle event. A value of 0 terminates Idle detection.

Return value

Integer. Returns 1 if it starts the timer, and -1 if it cannot start the timer or $n$ is 0 and the timer has not been started. Note that when the timer has been started and you change $n$, Idle does not start a new timer; it resets the current timer interval to the new number of seconds. If $n$ is null, Idle returns null. The return value is usually not used.

Usage

Use Idle to shut off or restart an application when there is no user activity. This is often done for security reasons.

Idle starts a timer after each user activity (such as a keystroke or a mouse click), and after $n$ seconds of inactivity it triggers an Idle event. The Idle event script for an application typically closes some windows, logs off the database, and exits the application or calls the Restart function.

The timer is reset when any of the following activities occur:

- A mouse movement or mouse click in any window of the application
- Any keyboard activity when a window of the PowerBuilder application is current
- A mouse click or any mouse movement over the icon when a PowerBuilder application is minimized
- Any keyboard activity when the PowerBuilder application is minimized and is current (its name is highlighted)
- Any retrieval on a visible DataWindow that causes the edit control to be painted

Tip

To capture movement, write script in the MouseMove or Key events of the window or sheet. (Keyboard activity does not trigger MouseMove events.) Disable the DataWindow control and tab ordering during iterative retrieves so the Idle timer is not reset.
ImpersonateClient

Examples
This statement sends an Idle event after five minutes of inactivity:

```
Idle(300)
```

This statement turns off idle detection:

```
Idle(0)
```

This example shows how to use the Idle event to stop the application and restart it after two minutes of inactivity. This is often used for computers that provide information in a public place.

Include this statement in the script for the application’s Open event:

```
Idle(120) // Sends an Idle event after 2 minutes.
```

Include these statements in the script for the application’s Idle event to terminate the application and then restart it:

```
// Statements to set the database to the desired state
...
Restart() // Restarts the application
```

See also
Restart
Timer

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
```
transactionserver.ImpersonateClient ( )
```

<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
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:

```powerScript
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**

```
graphname.ImportClipboard( { { importtype }, { startrow, endrow, startColumn } } )
```
### ImportClipboard

**Argument** | **Description**
---|---
importtype (optional) | An enumerated value of the SaveAsType DataWindow constant. Valid type arguments for ImportClipboard are:
- Text!
- CSV!
- XML!
If you want to generate an XML trace file, the XML! argument is required.

graphname | The name of the graph control to which you want to copy data from the clipboard.

startrow (optional) | The number of the first detail row in the clipboard that you want to copy. The default is 1.

For default XML import, if startrow is supplied, the first \( N \) \((\text{startrow} - 1)\) elements are skipped, where \( N \) is the DataWindow row size.

For template XML import, if startrow is supplied, the first \((\text{startrow} - 1)\) occurrences of the repetitive row mapping defined in the template are skipped.

endrow (optional) | 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 endrow 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 endrow is supplied, import stops after endrow occurrences of the repetitive row mapping defined in the template have been imported.

startcolumn (optional) | The number of the first column in the clipboard that you want to copy. The default is 1.

For default XML import, if startcolumn is supplied, import skips the first \((\text{startcolumn} - 1)\) elements in each row.

This argument has no effect on template XML import.

### 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
-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

gremployee:

```
gremployee.ImportClipboard()
```
This statement copies the data from the clipboard starting with row 2 column 3 and copying to row 30 column 5 to the graph `gr_employee`:

```
gr_employee.ImportClipboard(2, 30, 3)
```

### 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**

```
graphname.ImportFile ( { importtype }, filename {, startrow {, endrow {, 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></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: Text! CSV! XML! DBase2! 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>
 CHAPTER 10  PowerScript Functions

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startrow (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 startrow is supplied, the first $N$ (startrow - 1) elements are skipped, where $N$ is the DataWindow row size. For template XML import, if startrow is supplied, the first (startrow - 1) occurrences of the repetitive row mapping defined in the template are skipped.</td>
</tr>
<tr>
<td>endrow (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 endrow is supplied, import stops when $N \times$ endrow elements have been imported, where $N$ is the DataWindow row size. For template XML import, if endrow is supplied, import stops after endrow occurrences of the repetitive row mapping defined in the template have been imported.</td>
</tr>
<tr>
<td>startcolumn (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 startcolumn 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>

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)
-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 &quot;Ramakrishnan&quot;</td>
<td>Bernice &quot;Ramakrishnan&quot;</td>
</tr>
<tr>
<td>&quot;Mary&quot; TAB &quot;Li&quot;</td>
<td>Empty cells</td>
</tr>
<tr>
<td>&quot;Mich&quot;ael TAB &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:

```powerhouse
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`:

```powerhouse
gr_employee.ImportFile("D:\EMPLOYEE.TXT", 2, 30, 3)
```

The following statements are equivalent. Both import the contents of the XML file named `myxmldata`:

```powerhouse
gr_control.ImportFile(mymxmldata.xml)
gr_control.ImportFile(XML!, myxmxmldata)
```

This example causes PowerBuilder to display the Specify Import File dialog box:

```powerhouse
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 } } } )
```

<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 string.</td>
</tr>
<tr>
<td><code>importtype</code> (optional)</td>
<td>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.</td>
</tr>
<tr>
<td><code>string</code></td>
<td>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).</td>
</tr>
<tr>
<td><code>startrow</code> (optional)</td>
<td>The number of the first detail row in the string 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>
</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.
You can add data to more than one series by specifying different series names in the first column.

Examples

These statements copy the data from the string ls_Text starting with row 2 column 3 and ending with row 30 column 5 to the graph gr_employee:

```powerbuilder
string ls_Text
ls_Text = . . .
gr_employee.ImportString(ls_Text, 2, 30, 3)
```

The following script stores data for two series in the string ls_gr and imports the data into the graph gr_custbalance. The categories in the data are A, B, and C:

```powerbuilder
string ls_gr

ls_gr = "series1-tA-t12-r-n"
ls_gr = ls_gr + "series1-tB-t13-r-n"
ls_gr = ls_gr + "series1-tC-t14-r-n"
ls_gr = ls_gr + "series2-tA-t15-r-n"
ls_gr = ls_gr + "series2-tB-t14-r-n"
ls_gr = ls_gr + "series2-tC-t12.5-r-n"

gr_custbalance.ImportString(ls_gr, 1)
```

See also

ImportClipboard
ImportFile

**Description**

Provides a list of the callers of a routine included in a performance analysis model.

**Applies to**

ProfileRoutine object

**Syntax**

`instancename.IncomingCallList ( list, aggregateduplicateroutincalls )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instancename</code></td>
<td>Instance name of the ProfileRoutine object.</td>
</tr>
<tr>
<td><code>list</code></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>
CHAPTER 10  PowerScript Functions

Return value
ErrorReturn. Returns one of the following values:

- Success! – The function succeeded
- ModelNotExistsError! – The model does not exist

Usage
Use this function to extract a list of the callers of a routine included in a performance analysis model. Each caller is defined as a ProfileCall object and provides the called routine and the calling routine, the number of times the call was made, and the elapsed time. The callers are listed in no particular order.

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

The aggregateduplicateroutinecalls argument indicates whether duplicate routine calls will result in the creation of a single or of multiple ProfileCall objects. This argument has no effect unless line tracing is enabled and a calling routine calls the current routine from more than one line. If aggregateduplicateroutinecalls is true, a new ProfileCall object is created that aggregates all calls from the calling routine to the current routine. If aggregateduplicateroutinecalls is false, multiple ProfileCall objects are returned, one for each line from which the calling routine called the called routine.

Examples
This example gets a list of the routines included in a performance analysis model and then gets a list of the routines that called each routine:

```powerlanguage
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>jaguarorb</td>
<td>An instance of JaguarORB.</td>
</tr>
<tr>
<td>options</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>CORBACurrent</td>
<td>Reference to the CORBACurrent service instance.</td>
</tr>
<tr>
<td>connection</td>
<td>The name of the Connection object for which a connection has already been established to a valid EAServer host. Either connection or URL is required if the Init function is called by a client.</td>
</tr>
<tr>
<td>URL</td>
<td>String. The name of a URL that identifies a valid EAServer host. Either connection or URL 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:

```java
// 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:

```java
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:

```powerscript
li_rc = corbcurr.init( "iiop://localhost:9000" )
IF li_rc <> 0 THEN
   // handle the error
END IF
```

See also:
- `BeginTransaction`
- `CommitTransaction`
- `GetContextService`
- `GetStatus`
- `GetTransactionName`
- `ResumeTransaction`
- `RollbackOnly`
- `RollbackTransaction`
- `SetTimeout`
- `SuspendTransaction`

**InputFieldChangeData**

Description: Modifies the data value of input fields in a RichTextEdit control.

Applies to: RichTextEdit controls

Syntax:

```powerscript
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.
InputFieldChangeData

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`:

```powerbuilder
integer rtn
rtn = rte_1.InputFieldChangeData &
("instrument", lb_instruments.SelectedItem())

st_status.Text = String(rtn)
```

If the text in `rte_1` looks like this:

```
Dear {title} {lastname}:
We’re happy you have rented a {instrument} for your child. Please perform regular maintenance for the {instrument} as instructed by your child’s teacher. You can buy {instrument} supplies and instruction books at your local music stores.
```

Then after the user picks `trumpet` in the ListBox, the script inserts `trumpet` for every occurrence of the `{instrument}` field. The other fields are not affected:

```
Dear {title} {lastname}:
We’re happy you have rented a trumpet for your child. Please perform regular maintenance for the trumpet as instructed by your child’s teacher. You can buy trumpet supplies and instruction books at your local music stores.
```

See also

- InputFieldCurrentName
- InputFieldDeleteCurrent
- InputFieldGetData
- InputFieldInsert
- InputFieldLocate
- DataSource
**InputFieldCurrentName**

**Description**
Gets the name of the input field when the insertion point is in an input field in a RichTextEdit control.

**Applies to**
RichTextEdit controls

**Syntax**
```
rtename.InputFieldCurrentName()
```

**Return value**
String. Returns the name of the input field. If the insertion point is not in an input field or if an error occurs, it returns the empty string ("").

**Examples**
This example gets the name of the input field containing the insertion point:
```
string ls_inputname
ls_inputname = rte_1.InputFieldCurrentName()
```

**See also**
InputFieldChangeData
InputFieldDeleteCurrent
InputFieldGetData
InputFieldInsert
InputFieldLocate
DataSource
InputFieldDeleteCurrent

Description
Deletes the input field that is selected in a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
`rtename.InputFieldDeleteCurrent ( )`

Return value
Integer. Returns 1 if it succeeds and -1 if there is no input field at the insertion point, the input field is activated for editing, or an error occurs.

Usage
All the input fields that have the same name contain the same data but they can be deleted independently. If one of a group of input fields with the same name is deleted, the others are not affected. If all the input fields of the same name are deleted, the RichTextEdit control remembers the data from those input fields. It will use that data to initialize a new input field that has the same name as the deleted fields.

The input field must be the only selection. If other text is selected too, `InputFieldDeleteCurrent` fails. When an input field is the current and only selection, the highlight flashes.

InputFieldDeleteCurrent deletes only the current field. Other fields with the same name within the document are not affected. If the RichTextEdit control uses the `DataSource` function to share data with a DataWindow, the current field is deleted from all instances of the document.

Examples
This example deletes the input field containing the insertion point:

```powerbuilder
integer li_rtn
li_rtn = rte_1.InputFieldDeleteCurrent ()
```

See also
InputFieldChangeData
InputFieldGetData
InputFieldCurrentName
InputFieldInsert
InputFieldLocate
DataSource
**InputFieldGetData**

Description: Get the data in the specified input field in a RichTextEdit control.

Applies to: RichTextEdit controls

Syntax: 

```
rtename.InputFieldGetData ( inputfieldname )
```

**Argument** | **Description**
--- | ---
rtename | The name of the RichTextEdit control in which you want to get data from the selected input field
inputfieldname | A string whose value is the name of input field from which you want to get the data

Return value: String. The data in the input field. InputFieldGetData returns the empty string (""") if the field does not exist or an error occurs.

Examples: This example gets the data in the input field empname:

```
string ls_name
ls_name = rte_1.InputFieldGetData (empname)
```

See also: InputFieldChangeData, InputFieldCurrentName, InputFieldDeleteCurrent, InputFieldInsert, InputFieldLocate, DataSource

**InputFieldInsert**

Description: Inserts a named input field at the insertion point in a RichTextEdit control.

Applies to: RichTextEdit controls

Syntax: 

```
rtename.InputFieldInsert ( inputfieldname )
```

**Argument** | **Description**
--- | ---
rtename | The name of the RichTextEdit control in which you want to insert an input field
inputfieldname | A string whose value is the name of input field to be inserted. The name does not have to be unique

Return value: Integer. Returns 1 if it succeeds and -1 if an error occurs. If inputfieldname is null, InputFieldInsert returns null.
InputFieldLocate

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:

```PowerBuilder
st_status.Text = String( &
rte_l.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)
```

<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 locate an input field.</td>
</tr>
<tr>
<td>location</td>
<td>A value of the Location enumerated datatype that specifies the occurrence of the input field you want to locate. Values are:</td>
</tr>
<tr>
<td></td>
<td>• First! – The first occurrence in the document of <code>inputfieldname</code>, or if no name is specified, the first input field in the document</td>
</tr>
<tr>
<td></td>
<td>• Last! – The last occurrence in the document of <code>inputfieldname</code>, or if no name is specified, the last input field in the document</td>
</tr>
<tr>
<td></td>
<td>• Next! – The occurrence of <code>inputfieldname</code> that is after the insertion point, or if no name is specified, the next input field of any name after the insertion point</td>
</tr>
<tr>
<td></td>
<td>• Prior! – The occurrence of <code>inputfieldname</code> before the insertion point, or if no name is specified, the next input field of any name before the insertion point</td>
</tr>
</tbody>
</table>
Return value

String. Returns the name of the input field it located if it succeeds. InputFieldLocate returns an empty string if no matching input field is found or if an error occurs. If any argument is null, InputFieldLocate returns null.

Usage

There can be several input fields with the same name. Fields of a given name all have the same data value.

Examples

This example locates the next input field after the insertion point. If found, ls_name is set to the name of the input field:

```
string ls_name
ls_name = rte_1.InputFieldLocate(Next!)
```

This example locates the last input field in the document:

```
string ls_name
ls_name = rte_1.InputFieldLocate(Last!)
```

This example locates the last occurrence in the document of the input field named address. If found, ls_name is set to the value "address":

```
string ls_name
ls_name = rte_1.InputFieldLocate(Last!, "address")
```

See also

InputFieldChangeData
InputFieldCurrentName
InputFieldDeleteCurrent
InputFieldGetData
InputFieldInsert
DataSource
InsertCategory

Description
Inserts a category on the category axis of a graph at the specified position. Existing categories are renumbered to keep the category numbering sequential.

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

Syntax
`controlname.InsertCategory (categoryvalue, categorynumber)`

<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.

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:

```powerbuilder
gr_data.InsertCategory("Qty", 0)
```

is equivalent to:

```powerbuilder
gr_data.AddCategory("Qty")
```
Examples

These statements insert a category called Macs before the category named PCs in the graph `gr_product_data`:

```powerscript
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:

```powerscript
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><code>ole2control</code></td>
<td>The name of the OLE control in which you want to create a new object</td>
</tr>
<tr>
<td><code>classname</code></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`.
### InsertColumn

**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`:

```powerbuilder
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**

```
livewname.InsertColumn( index, label, alignment, width )
```

<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 want to insert a column.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>An integer whose value is the number of the column before which you are inserting a new column.</td>
</tr>
<tr>
<td><code>label</code></td>
<td>A string whose value is the name of the column you are inserting.</td>
</tr>
<tr>
<td><code>alignment</code></td>
<td>A value of the enumerated datatype Alignment specifying the alignment of the column you are inserting. Values are: Center!, Justify!, Left!, Right!</td>
</tr>
<tr>
<td><code>width</code></td>
<td>An integer whose value is the width of the column you are inserting, in PowerBuilder units.</td>
</tr>
</tbody>
</table>

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

This example inserts a column named Location, makes it right-aligned, and sets the column width to 300:

```powerScript
lv_list.InsertColumn(2, "Location", Right!, 300)
```

See also
AddColumn
DeleteColumn

**InsertData**

Inserts a data point in a series of a graph. You can specify the category for the data point or its position in the series. Does not apply to scatter graphs.

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.InsertData ( seriesnumber, datapoint, datavalue {, categoryvalue } )
```

<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 data into a series.</td>
</tr>
<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> (optional)</td>
<td>The category for this data value on the category axis. The datatype of categoryvalue 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.</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:

```plaintext
gr_data.InsertData(1, 0, 44, "Qty")
```

is equivalent to:

```plaintext
gr_data.ModifyData(1, 0, 44, "Qty")
```

and is also equivalent to:

```plaintext
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:
  ```plaintext
  gr_data.InsertData(1, 4, 44, "Qty")
  ```

- `ModifyData` changes the category label and the data value at position 4:
  ```plaintext
  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`:

```power_script
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:

```power_script
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**

```power_script
rtename.InsertDocument(filename, clearflag, filetype)
```
## InsertDocument

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtrename</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. <code>Filename</code> can include the file’s path.</td>
</tr>
</tbody>
</table>
| clearflag | A boolean value specifying whether the new file will replace the current contents of the control. Values are:  
  - `TRUE` – Replace the current contents with the file  
  - `false` – Insert the file into the existing contents at the insertion point |
| filetype (optional) | A value of the FileType enumerated datatype specifying the type of file being opened. Values are:  
  - `FileTypeRichText!` – (Default) The file being opened is in rich text format (RTF)  
  - `FileTypeText!` – The file being opened is plain ASCII text (TXT)  
  - `FileTypeHTML!` – The file being opened is in HTML format (HTM or HTML)  
  - `FileTypeDoc!` – The file being opened is in Microsoft Word format (DOC)  
  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 reads the file as plain text. |

### Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, InsertDocument returns null.

### Usage

When the control supports headers and 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!)
st_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
`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>

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

-1 File not found
-9 Other error

If any argument’s value is null, InsertFile returns null.

Usage
The contents of the specified file is embedded in the OLE object. There is no further link between the object in PowerBuilder and the file.

Examples
This example creates a new OLE object in the control ole_1. It is an Excel object and contains data from the spreadsheet EXPENSE.XLS:

```powerbuilder
integer result
result = ole_1.InsertFile("c:\xls\expense.xls")
```

See also
InsertClass
InsertObject
**InsertItem**

Inserts an item into a ListBox, DropDownListBox, ListView, or TreeView control.

**To insert an item into a**

<table>
<thead>
<tr>
<th><strong>ListBox or DropDownListBox control</strong></th>
<th><strong>Use</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax 1</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PictureBox or DropDownPictureListBox control</strong></td>
<td><strong>Syntax 2</strong></td>
</tr>
<tr>
<td><strong>ListView control when only the label and picture index need to be specified</strong></td>
<td><strong>Syntax 3</strong></td>
</tr>
<tr>
<td><strong>ListView control when more than the label and picture index need to be specified</strong></td>
<td><strong>Syntax 4</strong></td>
</tr>
<tr>
<td><strong>TreeView control when only the label and picture index need to be specified</strong></td>
<td><strong>Syntax 5</strong></td>
</tr>
<tr>
<td><strong>TreeView control when more than the label and picture index need to be specified</strong></td>
<td><strong>Syntax 6</strong></td>
</tr>
</tbody>
</table>

**Syntax 1**

**For ListBox and DropDownListBox controls**

Inserts an item into the list of values in a list box.

**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**

<table>
<thead>
<tr>
<th><strong>listboxname</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The name of the ListBox or DropDownListBox into which you want to insert an item</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>item</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A string whose value is the text of the item you want to insert</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>index</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The number of the item in the list before which you want to insert the item</td>
<td></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

*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 *lb_actions*:

```
lb_actions.InsertItem("Run Application", 5)
```

If the Sorted property is false, the statement above returns 5 (the previous item 5 becomes item 6). If the Sorted property is true, the list is sorted after the item is inserted and the function returns the index of the final position of the item.

If the ListBox *lb_Cities* has the following items in its list and its Sorted property is set to true, then the following example inserts Denver at the top, sorts the list, and sets *li_pos* to 4. If the ListBox’s Sorted property is false, then the statement inserts Denver at the top of the list and sets *li_pos* to 1. The list is:

- Albany
- Boston
- Chicago
- New York

The example code is:

```
string ls_City = "Denver"
integer li_pos
li_pos = lb_Cities.InsertItem(ls_City, 1)
```

See also

*AddItem*, *DeleteItem*, *FindItem*, *Reset*, *TotalItems*
**InsertItem**

**Argument** | **Description**
---|---
`listboxname` | The name of the PictureListBox or DropDownPictureListBox into which you want to insert an item
`item` | A string whose value is the text of the item you want to insert
`pictureindex` (optional) | An integer specifying the index of the picture you want to associate with the newly added item
`index` | The number of the item in the list before which you want to insert the item

Return value: Integer. Returns the final position of the item. Returns -1 if an error occurs. If any argument’s value is null, `InsertItem` returns null.

Usage:
If you 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)
```

See also:
AddItem
DeleteItem
FindItem
Reset
TotalItems

**Syntax 3**

**For ListView controls**

**Description**
Inserts an item into a ListView control.

**Applies to**
ListView controls

**Syntax**

```
listviewname.InsertItem ( index, label, pictureindex )
```
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>listviewname</td>
<td>The name of the ListView control into which you are inserting an item</td>
</tr>
<tr>
<td>index</td>
<td>An integer whose value is the index number of the item you are adding</td>
</tr>
<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:

```powerscript
listviewitem l_lvi
```
InsertItem

lv_list.GetItem(2, l_lvi)
lv_list.InsertItem(5, l_lvi)
lv_list.DeleteItem(2)

See also
AddItem
Syntax 5

For TreeView controls

 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.

Usage

Use this syntax to set just the label and picture index. Use the next syntax if you need to set additional properties for the item.

If the TreeView’s SortType property is set to a value other than Unsorted!, the inserted item is sorted with its siblings.

If you are inserting the first child of an item, use InsertItemLast or InsertItemFirst instead. Those functions 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:

```
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
InsertItem

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.

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:

```powershell
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

Inserts an item as the first child of a parent item.

## Syntax 1

### Description
Inserts an item as the first child of its parent.

### Applies to
TreeView controls

### Syntax
```
treeviewname.InsertItemFirst(handleparent, label, pictureindex)
```

### Argument Description

<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 specify an item as the first child of its parent.</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 of the item you want to specify as the first child of its parent.</td>
</tr>
<tr>
<td><code>pictureindex</code></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`:

```power_script
long ll_lev1, ll_lev2, ll_lev3, ll_lev4
int index

tv_list.PictureBox = 32
tv_list.PictureBox = 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)
```

---

**CHAPTER 10  PowerScript Functions**

**InsertItemFirst**

Inserts an item as the first child of a parent item.

<table>
<thead>
<tr>
<th>To insert an item as the first 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 the item label and picture index</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

For TreeView controls

Inserts an item as the first child of its parent.

**Syntax 1**

**Description**
Inserts an item as the first child of its parent.

**Applies to**
TreeView controls

**Syntax**
```
treeviewname.InsertItemFirst(handleparent, label, pictureindex)
```

**Argument Description**

<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 specify an item as the first child of its parent.</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 of the item you want to specify as the first child of its parent.</td>
</tr>
<tr>
<td><code>pictureindex</code></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`:

```power_script
long ll_lev1, ll_lev2, ll_lev3, ll_lev4
int index

tv_list.PictureBox = 32
tv_list.PictureBox = 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)
```
**InsertItemFirst**

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><code>treeviewname</code></td>
<td>The TreeView control in which you want to specify an item as the first child of its parent.</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>item</code></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!, 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:

```powerbuilder
long   ll_handle, ll_roothandle
treeviewitem  l_tvi
ll_handle = tv_list.FindItem(CurrentTreeItem!, 0)
ll_roothandle = tv_list.FindItem(RootTreeItem!, 0)
tv_list.GetItem(ll_handle , l_tvi)

tv_list.InsertItemFirst(ll_roothandle, l_tvi)
```

See also
InsertItem
InsertItemLast
InsertItemSort
InsertItemLast

Inserts an item as the last child of a parent item.

<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><code>treeviewname</code></td>
<td>The TreeView control in which you want to specify an item as the last child of its parent.</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 of the item you want to specify as the last child of its parent.</td>
</tr>
<tr>
<td><code>pictureindex</code></td>
<td>The picture index for the item you want to specify as the last 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.

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:

```powercript
long    11_lev1, 11_lev2, 11_lev3, 11_lev4
int     index

tv_list.PictureHeight = 32
tv_list.PictureWidth = 32
```
**InsertItemLast**

1. `ll_lev1 = tv_list.InsertItemLast(0, "Composers", 1)`
2. `ll_lev2 = tv_list.InsertItemLast(ll_lev1, "Beethoven", 2)`
3. `ll_lev3 = tv_list.InsertItemLast(ll_lev2, "Symphonies", 3)`
4. FOR `index` = 1 to 9
   - `ll_lev4 = tv_list.InsertItemSort(ll_lev3, "Symphony # " String(index), 4)`
5. `NEXT`
6. `tv_list.ExpandItem(ll_lev3)`
7. `tv_list.ExpandItem(ll_lev4)`

See also
- InsertItem
- InsertItemFirst
- InsertItemSort

**Syntax 2** For TreeView controls

**Description**
Inserts an item as the last child of its parent.

**Applies to**
TreeView controls

**Syntax**

```
treeviewname.InsertItemLast(handleparent, item)
```

<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>
<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!, 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)
```
```powerScript
ll_roothandle = tv_list.FindItem(RootTreeItem!, 0)
tv_list.GetItem(ll_handle, l_tvi)

tv_list.InsertItemLast(ll_roothandle, l_tvi)
```

See also
- InsertItem
- InsertItemFirst
- InsertItemSort

---

### InsertItemSort

Inserts a child item in sorted order under the parent item.

**To insert an item in sorted order**

<table>
<thead>
<tr>
<th>Description</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 )
```

**Argument**  | **Description**
---            | ---
`treeviewname` | The TreeView control in which you want to insert and sort an item as a child of its parent, according to its label.
`handleparent` | The handle of the item that will be the inserted item’s parent. To insert the item at the first level, specify 0.
`label`        | The label by which you want to sort the item as a child of its parent.
`pictureindex` | The picture index for the item you want to sort as a child of its parent, according to its label.

**Return value**

`Long`. Returns the handle of the item inserted if it succeeds and -1 if an error occurs.
**InsertItemSort**

**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:

```powershell
long ll_lev1, ll_lev2, ll_lev3, ll_lev4
int index

    tv_list.PictureHeight = 32
    tv_list.PictureWidth = 32

    ll_lev1 = tv_list.InsertItemLast(0, "Composers", 1)
    ll_lev2 = tv_list.InsertItemLast(ll_lev1, & "Beethoven", 2)
    ll_lev3 = tv_list.InsertItemLast(ll_lev2, & "Symphonies", 3)
    FOR index = 1 to 9
        ll_lev4 = tv_list.InsertItemSort(ll_lev3, & "Symphony # " + String(index), 4)
    NEXT

    tv_list.ExpandItem(ll_lev3)
    tv_list.ExpandItem(ll_lev4)
```

**See also**

InsertItem
InsertItemLast
InsertItemFirst

**Syntax 2**

**For TreeView controls**

**Description**

Inserts an item in sorted order, if possible.

**Applies to**

TreeView controls

**Syntax**

```powershell
treeviewname.InsertItemSort ( 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 sort an item 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 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**

```powerscript
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>
**InsertPicture**

**Description**
Inserts a bitmap at the insertion point in a RichTextEdit control.

**Applies to**
RichTextEdit controls

**Syntax**
\[ \text{rtename}.\text{InsertPicture} ( \text{filename} ) \]

**Argument** | **Description**
--- | ---
\text{rtename} | The name of the RichTextEdit control in which you want to insert a picture
\text{filename} | A string whose value is the name of the file that contains the bitmap

**Return value**
Integer. Returns 1 if it succeeds and -1 if an error occurs. If \text{filename} is null, \text{InsertPicture} returns null.

**Usage**
If there is a selection, \text{InsertPicture} inserts the bitmap at the beginning of the selection. The bitmap and the selection remain selected.

**Examples**
This example inserts a BMP file at the insertion point in the RichTextEdit control \text{rte}_1:

\begin{verbatim}
integer li_rtn
li_rtn = rte_1.InsertPicture("c:\windows\earth.bmp")
\end{verbatim}

**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
controlname.InsertSeries ( seriesname, 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 to insert a series.</td>
</tr>
<tr>
<td>seriesname</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>seriesnumber</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:

```powerScript
gr_data.InsertSeries("Costs", 0)
```

is equivalent to:

```powerScript
gr_data.AddSeries("Costs")
```

Examples
These statements insert a series before the series named Income in the graph `gr_product_data`:

```powerScript
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**
```plaintext
Int( n )
```

**Return value**
Integer. Returns the largest whole number less than or equal to `n`. If `n` is too small or too large to be represented as an integer, `Int` returns 0. If `n` is null, `Int` returns null.

**Usage**
When the result for `Int` would be smaller than -32768 or larger than 32767, `Int` returns 0 because the result cannot be represented as an integer.

**Examples**
These statements return 3.0:
```plaintext
Int(3.2)
Int(3.8)
```

The following statements return -4.0:
```plaintext
Int(-3.2)
Int(-3.8)
```

These statements remove the decimal portion of the variable and store the resulting integer in `li_nbr`:
```plaintext
integer li_nbr
li_nbr = Int(3.2) // li_nbr = 3
```

**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**
```plaintext
Integer ( stringorblob )
```
Return value

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:

\[
\text{Integer}("24")
\]

This statement returns the contents of the SingleLineEdit sle_Age as an integer:

\[
\text{Integer}(\text{sle\_Age}.\text{Text})
\]

This statement returns 0:

\[
\text{Integer}("3ABC") // 3ABC is not a number.
\]

This example checks whether the text of sle_data is a number before converting, which is necessary if the user might legitimately enter 0:

\[
\begin{align*}
\text{integer} & \quad \text{li\_new\_data} \\
\text{IF} & \quad \text{IsNumber(sle\_data}.\text{Text}) \text{ THEN} \\
& \quad \text{li\_new\_data} = \text{Integer(sle\_data}.\text{Text}) \\
\text{ELSE} & \quad \text{SetNull(li\_new\_data)} \\
\end{align*}
\]

After assigning blob data from the database to lb_blob, this example obtains the integer value stored at position 20 in the blob:

\[
\begin{align*}
\text{integer} & \quad i \\
i & = \text{Integer(\text{BlobMid(lb\_blob, 20, 2))}
\end{align*}
\]

See also

Double
Dec
IsNumber
Long
Real
Integer method for DataWindows in the DataWindow Reference or the online Help
InternetData

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
```
servicereference.InternetData ( data )
```

<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>data</td>
<td>Blob containing the complete data requested by a GetURL or PostURL function</td>
</tr>
</tbody>
</table>

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:
```
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

\[ \textbf{IntHigh} \ ( \textit{long} ) \]

<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:

```plaintext
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

\[ \textbf{IntLow} \ ( \textit{long} ) \]

<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:

```plaintext
integer nLow, nHigh
long LValue = 12345
nLow = IntLow(LValue) // The Low Integer is 12345.
nHigh = IntHigh(LValue) // The High Integer is 0.
```
**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**
```javascript
activexcontrol.InvokePBFunction(name, numarguments, arguments)
```

<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, this 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>
<tr>
<td>name</td>
<td>String specifying the name of the user-defined window function. This argument is passed by reference.</td>
</tr>
<tr>
<td>numarguments</td>
<td>Integer specifying the number of elements in the arguments array. The default is zero.</td>
</tr>
<tr>
<td>arguments</td>
<td>Variant array containing function arguments. In PowerBuilder, Variant maps to the Any datatype. This argument is passed by reference.</td>
</tr>
</tbody>
</table>

If you specify this argument, you must also specify `numarguments`. If you do not specify this argument and the function contains arguments, populate the argument list by calling the `SetArgElement` function once for each argument.

JavaScript cannot use this argument.

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

**Usage**
Call this function to invoke a user-defined window function in the child window contained in a PowerBuilder window ActiveX control.

To check the PowerBuilder function’s return value, call the `GetLastReturn` function.

JavaScript cannot use the `arguments` argument.

**Examples**
This JavaScript example calls the `InvokePBFunction` function:
```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";
rcd = 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

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)

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:

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

\[
\text{oleobject}.\text{IsAlive}() \\
\]

Return value
Boolean. Returns \text{true} if the server object appears to be running and \text{false} if it is dead.

Usage
Use the \text{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 \text{IsAlive} returns \text{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 \text{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:

\begin{verbatim}
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
\end{verbatim}
**IsAllArabic**

**Description**
Tests whether a particular string is composed entirely of Arabic characters.

**Syntax**

```powerbuilder
IsAllArabic( 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 is composed entirely of Arabic characters</td>
</tr>
</tbody>
</table>

**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:

```powerbuilder
IsAllArabic( sle_name.Text )
```

**See also**

- IsAnyArabic
- IsArabic
- IsArabicAndNumbers
- Reverse

---

**IsAllHebrew**

**Description**
Tests whether a particular string is composed entirely of Hebrew characters.

**Syntax**

```powerbuilder
IsAllHebrew( 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 is composed entirely of Hebrew characters</td>
</tr>
</tbody>
</table>

**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><code>string</code></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 )

<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 Hebrew character</td>
</tr>
</tbody>
</table>

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 )

<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 Arabic character.</td>
</tr>
</tbody>
</table>

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:

\[ \text{IsArabic}(\text{sle}_\text{name}.\text{Text}) \]

See also

IsAllArabic
IsAnyArabic
IsArabicAndNumbers
Reverse

**IsArabicAndNumbers**

Description

Tests whether a particular string is composed entirely of Arabic characters or numbers.

Syntax

\[
\text{IsArabicAndNumbers}(\text{string})
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></td>
<td>A string whose value you want to test to find out if it is composed entirely of Arabic characters or numbers</td>
</tr>
</tbody>
</table>

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:

\[ \text{IsArabicAndNumbers}(\text{sle}_\text{name}.\text{Text}) \]

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**
```powerbuilder
transactionserver.IsCallerInRole ( role )
```

**Argument** | **Description**
--- | ---
transactionserver | Reference to the TransactionServer service instance
role | A string expression containing the name of a role

**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.
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:

```powershell
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
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:

```powershell
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)
```

**Argument** | **Description**
--- | ---
`datevalue` | A string whose value you want to test to determine whether it is a valid date

**Return value**
Boolean. Returns true if `datevalue` is a valid date and false if it is not. If `datevalue` is null, IsDate returns null.

**Usage**
You can use IsDate to test whether a user-entered date is valid before you convert it to a date datatype. To convert a value into a date value, use the Date function.

**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)
```

**Argument** | **Description**
--- | ---
`character` | A character or string whose value you want to test to find out if it is an Hebrew character
<table>
<thead>
<tr>
<th>Return value</th>
<th>Boolean. Returns true if character is an Hebrew character and false if it is not.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>If you are not running a version of Windows that supports right-to-left languages, IsHebrew is set to false.</td>
</tr>
<tr>
<td>Examples</td>
<td>Under a version of Windows that supports right-to-left languages, this statement returns true if the SingleLineEdit <code>sle_name</code> begins with a Hebrew character:</td>
</tr>
<tr>
<td></td>
<td><code>IsHebrew(sle_name.Text)</code></td>
</tr>
<tr>
<td>See also</td>
<td>IsAllHebrew</td>
</tr>
<tr>
<td></td>
<td>IsAnyHebrew</td>
</tr>
<tr>
<td></td>
<td>IsHebrewAndNumbers</td>
</tr>
<tr>
<td></td>
<td>Reverse</td>
</tr>
</tbody>
</table>

**IsHebrewAndNumbers**

**Description**
Tests whether a particular string is composed entirely of Hebrew characters and numbers.

**Syntax**

`IsHebrewAndNumbers (string)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string</code></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>

<table>
<thead>
<tr>
<th>Return value</th>
<th>Boolean. Returns true if <code>string</code> is composed entirely of Hebrew characters and numbers and false if it is not.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>If you are not running a version of Windows that supports right-to-left languages, IsHebrewAndNumbers is set to false.</td>
</tr>
<tr>
<td>Examples</td>
<td>Under a version of Windows that supports right-to-left languages, this statement returns true if the SingleLineEdit <code>sle_name</code> is composed entirely of Hebrew characters and numbers:</td>
</tr>
<tr>
<td></td>
<td><code>IsHebrewAndNumbers(sle_name.Text)</code></td>
</tr>
<tr>
<td>See also</td>
<td>IsAllHebrew</td>
</tr>
<tr>
<td></td>
<td>IsAnyHebrew</td>
</tr>
<tr>
<td></td>
<td>IsHebrew</td>
</tr>
<tr>
<td></td>
<td>Reverse</td>
</tr>
</tbody>
</table>
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 ( )

<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 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 )
```

**Argument** | **Description**  
--- | ---  
`any` | A variable or expression that you want to test to determine whether its value is null  

**Return value**
Boolean. Returns true if `any` is null and false if it is not.

**Usage**
Use `IsNull` to test whether a user-entered value or a value retrieved from the database is null. `IsNull` works for all datatypes but does not work for arrays.

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.

**Examples**
These statements set `lb_test` to true:

```plaintext
integer a, b
boolean lb_test
SetNull(b)
lb_test = IsNull(a + b)
```

**See also**
- `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.

Examples
This statement returns true:

IsNumber ("32.65")

This statement returns false:

IsNumber ("A16")

If the SingleLineEdit sle_Age contains 32, these statements store 32 in li_YearsOld:

integer li_YearsOld
IF IsNumber (sle_Age.Text) THEN
   li_YearsOld = Integer (sle_Age.Text)
END IF

See also
Double
Dec
Integer
Long
Real
IsNumber 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**
```
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:
```
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>transactionserver</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:

```power
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**

```powerbuilder
IsTime ( timevalue )
```

**Argument** | **Description**
--- | ---
`timevalue` | A string whose value you want to test to determine whether it is a valid time

**Return value**
Boolean. Returns true if `timevalue` is a valid time and false if it is not. If `timevalue` is null, `IsTime` returns null.

**Usage**
Use `IsTime` to test to whether a value a user enters in an edit control is a valid time.

To convert a string to an time value, use the `Time` function.

**Examples**

This statement returns true:

```powerbuilder
IsTime("8:00:00 am")
```

This statement returns false:

```powerbuilder
IsTime("25:00")
```

If the SingleLineEdit `sle_EndTime` contains 4:15 these statements store 04:15:00 in `lt_QuitTime`:

```powerbuilder
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();

Argument | Description
--- | ---
transactionserver | Reference to the TransactionServer service instance

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.

```powerShell
// 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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keycode</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.

---

*PowerScript Reference Volume 2*
<table>
<thead>
<tr>
<th>Table 10-5: KeyCode values for keyboard keys</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of key</strong></td>
</tr>
<tr>
<td><strong>KeyCode values and descriptions</strong></td>
</tr>
<tr>
<td>Mouse buttons</td>
</tr>
<tr>
<td>KeyLeftButton! Left mouse button</td>
</tr>
<tr>
<td>KeyMiddleButton! Middle mouse button</td>
</tr>
<tr>
<td>KeyRightButton! Right mouse button</td>
</tr>
<tr>
<td>Letters</td>
</tr>
<tr>
<td>KeyA! - KeyZ! A - Z, uppercase or lowercase</td>
</tr>
<tr>
<td>Other symbols</td>
</tr>
<tr>
<td>KeyQuote! ‘ and &quot;</td>
</tr>
<tr>
<td>KeyEqual! = and +</td>
</tr>
<tr>
<td>KeyComma! , and &lt;</td>
</tr>
<tr>
<td>KeyDash! - and _</td>
</tr>
<tr>
<td>KeyPeriod! . and &gt;</td>
</tr>
<tr>
<td>KeySlash! / and ?</td>
</tr>
<tr>
<td>KeyBackQuote! ` and ~</td>
</tr>
<tr>
<td>KeyLeftBracket! [ and {</td>
</tr>
<tr>
<td>KeyBackSlash! \ and</td>
</tr>
<tr>
<td>KeyRightBracket! ] and }</td>
</tr>
<tr>
<td>KeySemiColon! ; and :</td>
</tr>
<tr>
<td>Non-printing characters</td>
</tr>
<tr>
<td>KeyBack! Backspace</td>
</tr>
<tr>
<td>KeyTab!</td>
</tr>
<tr>
<td>KeyEnter!</td>
</tr>
<tr>
<td>KeySpaceBar!</td>
</tr>
<tr>
<td>Function keys</td>
</tr>
<tr>
<td>KeyF1! - KeyF12! Function keys F1 to F12</td>
</tr>
<tr>
<td>Control keys</td>
</tr>
<tr>
<td>KeyShift!</td>
</tr>
<tr>
<td>KeyControl!</td>
</tr>
<tr>
<td>KeyAlt!</td>
</tr>
<tr>
<td>KeyPause!</td>
</tr>
<tr>
<td>KeyCapsLock!</td>
</tr>
<tr>
<td>KeyEscape!</td>
</tr>
<tr>
<td>KeyPrintScreen!</td>
</tr>
<tr>
<td>KeyInsert!</td>
</tr>
<tr>
<td>KeyDelete!</td>
</tr>
<tr>
<td>Navigation keys</td>
</tr>
<tr>
<td>KeyPageUp!</td>
</tr>
<tr>
<td>KeyPageDown!</td>
</tr>
<tr>
<td>KeyEnd!</td>
</tr>
<tr>
<td>KeyHome!</td>
</tr>
<tr>
<td>KeyLeftArrow!</td>
</tr>
<tr>
<td>KeyUpArrow!</td>
</tr>
<tr>
<td>KeyRightArrow!</td>
</tr>
<tr>
<td>KeyDownArrow!</td>
</tr>
</tbody>
</table>
The following code checks whether the user pressed the F1 key or the Ctrl key and executes some statements appropriate to the key pressed:

```power-script
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:

```power-script
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):

```power-script
IF KeyDown(65) THEN ...
```

This statement tests whether the user pressed the Shift key and the A key:

```power-script
IF KeyDown(65) AND KeyDown(KeyShift!) THEN ...
```

This statement in a Clicked event checks whether the Shift is also pressed:

```power-script
IF KeyDown(KeyShift!) THEN ...
```

### Type of key

<table>
<thead>
<tr>
<th>Type of key</th>
<th>KeyCode values and descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric and symbol</td>
<td>Key0! 0 and )</td>
</tr>
<tr>
<td>keys</td>
<td>Key1! 1 and !</td>
</tr>
<tr>
<td></td>
<td>Key2! 2 and @</td>
</tr>
<tr>
<td></td>
<td>Key3! 3 and #</td>
</tr>
<tr>
<td></td>
<td>Key4! 4 and $</td>
</tr>
<tr>
<td></td>
<td>Key5! 5 and %</td>
</tr>
<tr>
<td></td>
<td>Key6! 6 and ^</td>
</tr>
<tr>
<td></td>
<td>Key7! 7 and &amp;</td>
</tr>
<tr>
<td></td>
<td>Key8! 8 and *</td>
</tr>
<tr>
<td></td>
<td>Key9! 9 and (</td>
</tr>
<tr>
<td>Keypad numbers</td>
<td>KeyNumpad0! - KeyNumpad9! 0 - 9 on numeric keypad</td>
</tr>
<tr>
<td>Keypad symbols</td>
<td>KeyMultiply! * on numeric keypad</td>
</tr>
<tr>
<td></td>
<td>KeyAdd! + on numeric keypad</td>
</tr>
<tr>
<td></td>
<td>KeySubtract! - on numeric keypad</td>
</tr>
<tr>
<td></td>
<td>KeyDecimal! . on numeric keypad</td>
</tr>
<tr>
<td></td>
<td>KeyDivide! / on numeric keypad</td>
</tr>
<tr>
<td></td>
<td>KeyNumLock!</td>
</tr>
<tr>
<td></td>
<td>KeyScrollLock!</td>
</tr>
</tbody>
</table>

### 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:

```power-script
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:

```power-script
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):

```power-script
IF KeyDown(65) THEN ...
```

This statement tests whether the user pressed the Shift key and the A key:

```power-script
IF KeyDown(65) AND KeyDown(KeyShift!) THEN ...
```

This statement in a Clicked event checks whether the Shift is also pressed:

```power-script
IF KeyDown(KeyShift!) THEN ...
```
LastPos

Description
Finds the last position of a target string in a source string.

Syntax

\[
\text{LastPos} \left( \text{string1}, \text{string2} \left\{, \text{searchlength} \right\} \right)
\]

Argument | Description |
---|---|
\text{string1} | The string in which you want to find \text{string2}. |
\text{string2} | The string you want to find in \text{string1}. |
\text{searchlength} (optional) | A long that limits the search to the leftmost \text{searchlength} characters of the source string \text{string1}. The default is the entire string. |

Return value
Long. Returns a long whose value is the starting position of the last occurrence of \text{string2} in \text{string1} within the characters specified in \text{searchlength}. If \text{string2} is not found in \text{string1} or if \text{searchlength} is 0, \text{LastPos} returns 0. If any argument’s value is null, \text{LastPos} returns null.

Usage
The \text{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:

\[
\text{LastPos} \left( \text{"BABE RUTH"}, \text{"RU"} \right)
\]
This statement returns 3:

\[
\text{LastPos} \left( \text{"BABE RUTH"}, \text{"B"} \right)
\]
This statement returns 0, because the case does not match:

\[
\text{LastPos} \left( \text{"BABE RUTH"}, \text{"be"} \right)
\]
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:

\[
\text{LastPos} \left( \text{"BABE RUTH"}, \text{"RU"}, 4 \right)
\]
These statements change the text in the SingleLineEdit \text{sle_group}. The last instance of the text NY is changed to North East:

\[
\text{long place_nbr}
\text{place_nbr} = \text{LastPos} \left( \text{sle_group.Text}, \text{"NY"} \right)
\text{sle_group.SelectText} \left( \text{place_nbr}, 2 \right)
\text{sle_group.ReplaceText} \left( \text{"North East"} \right)
\]
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:

```powerbuilder
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:

```powerbuilder
// Tokenize the source string backwards
// Results in "pbsyc105.dll  powerbuilder
// shared  sybase  programs  c:

string sSource = &
    'c:\programs\sybase\shared\powerbuilder\pbsyc105.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
\[ \text{LeftA}(\text{string}, \text{n}) \]

<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
\[ \text{LeftW}(\text{string}, \text{n}) \]

LeftTrim

Description
Removes spaces from the beginning of a string.

Syntax
\[ \text{LeftTrim}(\text{string}) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string you want returned with leading spaces deleted</td>
</tr>
</tbody>
</table>

Return value
String. Returns a copy of \( \text{string} \) with leading spaces deleted if it succeeds and the empty string (""") if an error occurs. If \( \text{string} \) is null, LeftTrim returns null.
**LeftTrimW**

Examples

This statement returns RUTH:

```powerbuilder
LeftTrim(" RUTH")
```

These statements delete leading spaces from the text in the MultiLineEdit `mle_name` and store the result in `emp_name`:

```powerbuilder
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

```powerbuilder
LeftTrimW ( string )
```

---

**Len**

Description

Reports the length of a string or a blob.

Syntax

```powerbuilder
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:

```powerShell
Len(""")
```

These statements store in the variable `s_address_len` the length of the text in the `SingleLineEdit sle_address`:

```powerShell
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:

```powerShell
blob ib_blob
```

The sample code is:

```powerShell
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, `ib_blob` is initialized to the size 100 when it is declared. When you call `Len` for `ib_blob`, it always returns 100. This example uses `BlobEdit`, instead of `Blob`, to assign data to the blob because its size is already established. The declaration of the instance variable is:

```powerShell
blob{100} ib_blob
```

The sample code is:

```powerShell
long ll_len
ll_len = Len(ib_blob) // ll_len set to 100
BlobEdit(ib_blob, 1, "Test String")
ll_len = Len(ib_blob) // ll_len set to 100
```

See also

`Len` method for DataWindows in the `DataWindow Reference` or the online Help
**LenA**

**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.

**Syntax**

```
LenA (stringorblob)
```

**Argument** | **Description**
--- | ---
stringorblob | The string or blob for which you want the length in number of bytes

**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**
`LenA` replaces the functionality that `Len` had in DBCS environments in PowerBuilder 9.

In SBCS environments, `Len`, `LenW`, and `LenA` return the same results.

If you specify a size when you declare a blob, that is the size reported by `LenA`. If you do not specify a size for the blob, `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. `LenA` reports the length of the blob as the number of single-byte characters it can contain. `Len` and `LenW` report the size of the blob as the number of double-byte characters it can contain.

---

**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**

```
LenW (stringorblob)
```
Length

Reports the length in bytes of an open OLE stream.

**Len function**
To get the length of a string or blob, use the `Len` function.

**Applies to**
OLEStream objects

**Syntax**
```powerscript
olestream.Length(sizevar)
```

<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>sizevar</code></td>
<td>A long variable in which Length will store the size of <code>olestream</code></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 `MYSTUFF.OLE` and assigns it to the OLEStorage object `stg_stuff`. Then it opens the stream called `info` in `stg_stuff` and assigns it to the stream object `olestr_info`. Finally, it finds out the stream’s length and stores the value in the variable `info_len`.

The example does not check the function’s return values for success, but you should be sure to check the return values in your code:

```powerscript
boolean lb_memexists
OLEStorage stg_stuff
OLEStream olestr_info
long info_len

stg_stuff = CREATE oleStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

olestr_info.Open(stg_stuff, "info", &
    stgRead!, stgExclusive!)
olestr_info.Length(info_len)
```

**See also**
Open
Read
Seek
Write
LibraryCreate

Description
Creates an empty PowerBuilder library with optional comments.

Syntax
\[
\text{LibraryCreate}( \ libraryname, \ {, \ comments} )
\]

<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:

\[
\text{LibraryCreate}("c:\pb\dwTemp.pbl", \ "Temporary library for dynamic DataWindows")
\]

See also
LibraryDelete
LibraryDirectory
LibraryExport
LibraryImport
LibraryDelete

**Description**
Deletes a library file or, if you specify a DataWindow object, deletes the DataWindow object from the library.

**Syntax**
```
LibraryDelete ( libraryname {, objectname, objecttype } )
```

<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 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.</td>
</tr>
<tr>
<td>objectname</td>
<td>(optional) A string whose value is the name of the DataWindow object you want to delete from libraryname.</td>
</tr>
<tr>
<td>objecttype</td>
<td>(optional) A value of the LibImportType enumerated datatype identifying the type of object you want to delete. The only supported object type is ImportDataWindow!.</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, 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: 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:

```powershell
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
`LibraryDirectoryEx (libraryname, objecttype)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>libraryname</code></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><code>objecttype</code></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>
LibraryDirectoryEx

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
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:

```powerbuilder
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>
</tbody>
</table>
| objecttype   | A value of the LibExportType enumerated datatype identifying the type of objects you want to export:  
  - ExportApplication! – Application object  
  - ExportDataWindow! – DataWindow object  
  - ExportFunction! – Function object  
  - ExportMenu! – Menu object  
  - ExportPipeline! – Pipeline objects  
  - ExportProject! – Project objects  
  - ExportQuery! – Query objects  
  - ExportStructure! – Structure object  
  - ExportUserObject! – User objects  
  - ExportWindow! – Window object |

Return value
String. Returns the syntax of the object if it succeeds. The syntax is the same as the syntax returned when you export an object in the Library painter except that LibraryExport does not include an export header. Returns the empty string ("") if an error occurs. If any argument’s value is null, LibraryExport returns null.

Examples
These statements export the DataWindow object `dw_emp` from the library called `dwTemp` to a string named `ls_dwsyn` and then use it to create a DataWindow:
```
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</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:

```
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**

**Description**

Determines the number of lines in an edit control that allows multiple lines.

**Applies to**

RichTextEdit, MultiLineEdit, EditMask, and DataWindow controls

**Syntax**

`editname.LineCount()`

**Argument | Description**
--- | ---
`editname` | The name of the DataWindow control, EditMask, MultiLineEdit, or RichTextEdit for which you want the number of lines

**Return value**

Long. Returns the number of lines in `editname` if it succeeds and -1 if an error occurs. If `editname` is null, LineCount returns null.

**Usage**

LineCount counts each visible line, whether it was the result of wrapping or carriage returns.

When you call LineCount for a DataWindow, it reports the number of lines in the edit control over the current row and column. A user can enter multiple lines in a DataWindow column only if it has a text 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:

```
integer li_Count
li_Count = mle_Instructions.LineCount()
```

These statements display a MessageBox if fewer than two lines have been entered in the MultiLineEdit `mle_Address`:

```
integer li_Lines
li_Lines = mle_Address.LineCount()
IF li_Lines < 2 THEN
   MessageBox("Warning", "2 lines are required.")
END IF
```

**LineLength**

**Description**
Determines the length of the line containing the insertion point in an edit control.

**Applies to**
RichTextEdit, MultiLineEdit, and EditMask controls

**Syntax**
```
editname.LineLength ( )
```

<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_Contact and line 5 contains the text Select All, il_linelength is set to 10 (the length of line 5):

```powerscript
integer li_linelength
li_linelength = mle_Contact.LineLength()
```

See also
Position
SelectedLine
SelectedStart
TextLine

LineList

Description
Provides a list of the lines in a routine included in a performance analysis model.

Applies to
ProfileRoutine object

Syntax

```
iinstancetype.LineList ( list )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instancename</td>
<td>Instance name of the ProfileRoutine object.</td>
</tr>
<tr>
<td>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>
**LineList**

**Return value**  
ErrorReturn. Returns one of the following values:
- Success! – The function succeeded
- ModelNotExistsError! – The model does not exist

**Usage**  
Use this function to extract a list of the lines in a routine included in the performance analysis model. You must have previously created the performance analysis model from a trace file using the `BuildModel` function. Each line is defined as a ProfileLine object and provides the number of times the line was hit, any calls made from the line, and the time spent on the line and in any called functions. The lines are listed in numeric order.

Lines are not returned for database statements and objects. If line information was not logged in the trace file, lines are not returned.

**Examples**  
This example gets a list of the routines included in a performance analysis model and then gets a list of the lines in each routine:

```powerbuilder
Long ll_cnt
ProfileLine lproln_line[]

lpro_model.BuildModel()
lpro_model.RoutineList(iprort_list)

FOR ll_cnt = 1 TO UpperBound(iprort_list)
   iprort_list[ll_cnt].LineList(lproln_line)
      ...
NEXT
```

**See also**  
`BuildModel`
**LinkTo**

Establishes a link between an OLE control and a file or an item within the file.

**Syntax**

```powerscript
olecontrol.LinkTo ( filename {, sourceitem } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>olecontrol</strong></td>
<td>The name of the OLE control in which you want to insert a linked object.</td>
</tr>
<tr>
<td><strong>filename</strong></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><strong>sourceitem</strong> (optional)</td>
<td>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`:

```powerscript
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:

```powerscript
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**

```powerbuilder
inkpicname.LoadInk (t | b)
```

**Argument** | **Description**
--- | ---
`inkpicname` | The name of the InkPicture control into which you want to load ink.
`t` | A string containing the name and location of a file that contains the ink you want to load into the control.
`b` | The name of a blob passed by reference that contains the ink you want to load into the control.

**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:

```powerbuilder
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:

```powerbuilder
string ls_inkpath, ls_inkname
integer li_filenum
blob lblob_ink

GetFileOpenName("Select Ink File", ls_inkpath, &
ls_inkname)
li_filenum = FileOpen(ls_inkpath, StreamMode!)
If li_filenum <> 1 Then
    FileRead(li_filenum, lblob_ink)
    FileClose(li_filenum)
    ip_1.LoadInk(lblob_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:

```powershell
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:

```powershell
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

Log (n)

Argument | Description
---|---
\(n\) | The number for which you want the natural logarithm (base \(e\)). The value of \(n\) must be greater than 0.

Return value

Double. Returns the natural logarithm of \(n\). An execution error occurs if \(n\) is negative or zero. If \(n\) is null, Log returns null.

Inverse of Log

The inverse of the Log function is the Exp function.

Examples

This statement returns 2.302585092:

\[ \text{Log}(10) \]

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:

```plaintext
double a, b = Log(200)
a = 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
Wrote 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.
In the expression $10^n$ is the inverse of $\log_{10}(n)$. To obtain the value of $n$ in the equation $x = \log_{10}(n)$, use $n = 10^x$.

Examples

This statement returns 1:

``` powershell
LogTen(10)
```

The following statements both return 0:

``` powershell
LogTen(1)
LogTen(0)
```

This statement results in an execution error:

``` powershell
LogTen(-2)
```

After the following statements execute, the value of `a` is 200:

``` powershell
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

``` powershell
Long (lowword, highword)
```
Return value
Long. Returns the long if it succeeds and -1 if an error occurs. If any argument’s value is null, Long returns null.

Usage
Use Long for passing values to external C functions or specifying a value for the LongParm property of PowerBuilder’s Message object.

Examples
These statements convert the UnsignedIntegers $nLow$ and $nHigh$ into a long value:

```
UnsignedInt nLow  // Low integer 16 bits
UnsignedInt nHigh // High integer 16 bits
long LValue      // Long value 32 bits

nLow = 12345
nHigh = 0
LValue = Long(nLow, nHigh)
MessageBox("Long Value", Lvalue)
```

Syntax 2
For converting strings and blobs

Description
Converts a string whose value is a number into a long or obtains a long value stored in a blob.

Syntax
```
Long ( stringorblob )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringorblob</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. Stringorblob can also be an Any variable containing a string or blob.</td>
</tr>
</tbody>
</table>

_address_
LongLong

Examples

This statement returns 2167899876 as a long:

```
Long("2167899876")
```

After assigning blob data from the database to `lb_blob`, the following example
obtains the long value stored at position 20 in the blob:

```
long lb_num
lb_num = Long(BlobMid(lb_blob, 20, 4))
```

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

See also

Dec
Double
Integer
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

```
LongLong ( lowword, highword )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowword</td>
<td>An UnsignedLong to be the low word in the longlong</td>
</tr>
<tr>
<td>highword</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 \texttt{lLow} and \texttt{lHigh} into a long value:

\begin{verbatim}
UnsignedLong lLow    //Low long 32 bits
UnsignedLong lHigh   //High long 32 bits
longlong LLValue    //LongLong value 64 bits
\end{verbatim}

\begin{verbatim}
lLow = 1234567890
lHigh = 9876543210
LLValue = \textbf{LongLong}(lLow, lHigh)
MessageBox("LongLong Value", LLValue)
\end{verbatim}

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

\textbf{LongLong} ( \textit{stringorblob} )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{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. \textit{Stringorblob} can also be an \texttt{Any} variable containing a string or blob.</td>
</tr>
</tbody>
</table>

Return value

LongLong. Returns the value of \textit{stringorblob} as a longlong if it succeeds and 0 if \textit{stringorblob} is not a valid PowerScript number or if it is an incompatible datatype. If \textit{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 \textbf{IsNumber} function before calling the \textbf{LongLong} function.

Examples

This statement returns 216789987654321 as a longlong:

\begin{verbatim}
LongLong("216789987654321")
\end{verbatim}

After assigning blob data from the database to \texttt{lb_blob}, the following example obtains the longlong value stored at position 20 in the blob:

\begin{verbatim}
longlong llb_num
llb_num = \textbf{LongLong}(BlobMid(lb_blob, 20, 4))
\end{verbatim}

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)`

**Argument** | **Description**
---|---
`objname` | The name of the Connection object used to establish the connection or of an instance of the TransactionServer context object.
`objectvariable` | A global, instance, or local variable of the factory interface type.
`componentname` | 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, "mypackage/mycomponent").

**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:

```powerbuilder
// 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><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 type of the home interface to be created.</td>
</tr>
</tbody>
</table>
Lookup

**Argument** | **Description**
---|---
componentname | 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, "mypackage/mycomponent").
homeid | A string whose value is the name of the home interface to be created. This argument is optional

**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
```
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:

```powerbuilder
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
Lower

Description
Converts all the characters in a string to lowercase.

Syntax
Lower ( 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 lowercase letters</td>
</tr>
</tbody>
</table>

Return value
String. Returns string with uppercase letters changed to lowercase if it succeeds and the empty string ("") if an error occurs. If string is null, Lower returns null.

Examples
This statement returns babe ruth:

    Lower ("Babe Ruth")

See also
Upper

Lower method for DataWindows in the DataWindow Reference or the online Help

LowerBound

Description
Obtains the lower bound of a dimension of an array.

Syntax
LowerBound ( 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 lower bound of a dimension</td>
</tr>
<tr>
<td>n</td>
<td>The number of the dimension for which you want the lower bound. The default is 1</td>
</tr>
</tbody>
</table>

Return value
Long. Returns the lower bound of dimension n of array and -1 if n is greater than the number of dimensions of the array. If any argument’s value is null, LowerBound returns null.
LowerBound

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:

```powershell
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**

```
mailSession.mailAddress ( { 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 address the message.</td>
</tr>
<tr>
<td>mailmessage</td>
<td>A mailMessage structure containing information about the message. If you omit mailmessage, mailAddress displays an Address dialog box. (optional)</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:

```
mailSession mSes
mailReturnCode mRet
mailMessage mMsg
mailFileDescription mAttach
```
// Create a mail session
mSes = CREATE mailSession

// Log on to the session
mRet = mSes.mailLogon(mailNewSession!)
IF mRet <> mailReturnSuccess! THEN
    MessageBox("Mail", 'Logon failed.')
    RETURN
END IF
mMsg.AttachmentFile[1] = mAttach
mRet = mSes.mailAddress(mMsg)
IF mRet <> mailReturnSuccess! THEN
    MessageBox("Mail", 'Addressing failed.')
    RETURN
END IF

// Send the mail
mRet = mSes.mailSend(mMsg)
IF mRet <> mailReturnSuccess! THEN
    MessageBox("Mail", 'Sending mail failed.')
    RETURN
END IF
mSes.mailLogoff()
DESTROY mSes

See also
mailLogoff
mailLogon
mailResolveRecipient
mailSend
**mailDeleteMessage**

**Description**
Deletes a mail message from the user’s electronic mail inbox.

**Applies to**
mailSession object

**Syntax**
```
mailsession.mailDeleteMessage( messageid )
```

**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:

```powerscript
string sID
integer nRow
mailReturnCode mRet

nRow = GetClickedRow()
IF nRow > 0 THEN
    sID = GetItemString(nRow, "messageID")
    mRet = mSes.mailDeleteMessage(sID)
END IF
```

**See also**
mailGetMessages
mailLogon
mailGetMessages

Description
Populates the messageID array of a mailSession object with the message IDs in the user’s inbox.

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.

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`

### mailHandle

**Description**

Obtains the handle of a mailSession object.

**Applies to**

- `mailSession` object

**Syntax**

```powerbuilder
dailSession.mailHandle ()
```

**Argument** | **Description**
--- | ---
`mailSession` | A mailSession object identifying the session for which you want the handle

**Return value**

UnsignedLong. Returns the internal handle of the mail session object. If `mailSession` is null, `mailHandle` 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 `mailLogon` to establish a mail session.
Examples

This statement returns the handle of the current mail session:

```powerbuilder
current_session. `mailHandle`()
```
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
\texttt{mailsession.mailLogon ( \{ profile, password \} \{, logonoption \} )}

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:

```powerbuilder
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:

```powerbuilder
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**

```powerbuilder
mailsession.mailReadMessage ( messageId, mailmessage, readoption, mark )
```

<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>
CHAPTER 10  PowerScript Functions

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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readoption</td>
<td>A value of the mailReadOption enumerated datatype:</td>
</tr>
<tr>
<td></td>
<td>• mailEntireMessage! – Obtain header, text, and attachments</td>
</tr>
<tr>
<td></td>
<td>• mailEnvelopeOnly! – Obtain header information only</td>
</tr>
<tr>
<td></td>
<td>• mailBodyAsFile! – Obtain header, text, and attachments, and treat the message text as the first attachment, storing it in a temporary file</td>
</tr>
<tr>
<td></td>
<td>• mailSuppressAttachments! – Obtain header and text, but no attachments</td>
</tr>
<tr>
<td>mark</td>
<td>A boolean indicating whether you want to mark the message as read in the user’s inbox. Values are:</td>
</tr>
<tr>
<td></td>
<td>• TRUE – Mark the message as read</td>
</tr>
<tr>
<td></td>
<td>• FALSE – Do not mark the message as read</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
In this example, mail is displayed in a window with a DataWindow `dw_inbox` that lists mail messages and a MultiLineEdit `mle_note` that displays the message text. Assuming that the application has created the mailSession object `mSes` and successfully logged on, and that `dw_inbox` contains a list of mail items (sender, subject, postmark, and message ID); this script for the Clicked event for `dw_inbox` displays the text of the selected message in the MultiLineEdit `mle_note`:

```
integer nRow, nRet
string sMessageID
string sRet, sName

// Find out what Mail Item was selected
nRow = GetClickedRow()
IF nRow > 0 THEN
    // Get the message ID from the row
    sMessageID = GetItemString(nRow, 'MessageID')

    // Reread the message to obtain entire contents
    // because previously we read only the envelope
    nRet = 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})

Argument | Description
--- | ---
mailsession | A mailSession identifying the session in which you want to display the detail information for a recipient.
mailrecipient | A mailRecipient structure containing valid address information. Mailrecipient must contain a recipient identifier returned by mailAddress, mailResolveRecipient, or mailReadMessage.
allowupdates | (optional) A boolean indicating whether updates to the recipient’s name will be allowed. If the user does not have update privileges for the mail system, then allowupdates is ignored. The default is false.

Return value
mailReturnCode. Returns one of the following values:
- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!
- mailReturnUnknownRecipient!
- mailReturnUserAbort!

If any argument’s value is null, mailRecipientDetails returns null.

Usage
The effect of setting allowupdates to true depends on the mail system and the user’s privileges.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Examples
This example gets the message IDs from the user’s inbox and reads the first message. It then calls mailRecipientDetails to display address information for the first recipient. Recipient is an array of structures and a property of mailMessage. Each array element is one of the message’s recipients. The example does not check how many values there are in the message ID or recipient arrays and it assumes that the application has already created a mailSession object and logged on:

```powerScript
mailMessage msg
integer n
long c_row
```
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>mailSession</td>
<td>A mailSession object identifying the session in which you want to resolve the recipient.</td>
</tr>
<tr>
<td>recipient</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>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!
- mailReturnUserAbort!

Usage
Use mailResolveRecipient to verify that a name is a valid address in the mail system. The function reports mailReturnFailure! if the name is not found.
If you supply a mailRecipient structure, mailResolveRecipient fills the structure with valid address information when it resolves the address. If you supply a name as a string, mailResolveRecipient replaces the string’s value with the full user name as recognized by the mail system. An address specified as a string is adequate for users in the local mail system. If you are sending mail through gateways to other systems, you should obtain full address details in a mailRecipient structure.

If more than one address on the mail system matches the partial address information you supply to mailResolveRecipient, the mail system may display a dialog box allowing the user to choose the desired name.

If you supply a mailRecipient structure that already has address information, mailResolveRecipient 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.

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:

```powerScript
mailReturnCode mRet
string mname
mname = "Smith, J"
mRet = mSes.mailResolveRecipient(mname)
IF mRet = mailReturnSuccess! THEN
    MessageBox("Address", mname + " found.")
ELSEIF mRet = mailReturnFailure! THEN
    MessageBox("Address", "J Smith not found.")
ELSE
    MessageBox("Address", "Request not evaluated.")
END IF
```
In this example, `sle_to` contains the full or partial name of a mail recipient. This example assigns the name to a `mailRecipient` object and calls `mailResolveRecipient` to find the name and get address details. If the name is found, `mailRecipientDetails` displays the information and the full name is assigned to `sle_to`. The application has already created the `mailSession` object `mSes` and logged on:

```plaintext
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**

```plaintext
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 (&quot;&quot;&quot;).</td>
</tr>
<tr>
<td><code>mailmessage</code></td>
<td>A <code>mailMessage</code> structure containing the message being saved.</td>
</tr>
</tbody>
</table>
Return value

<table>
<thead>
<tr>
<th>mailReturnCode. Returns one of the following values:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailReturnSuccess!</td>
</tr>
<tr>
<td>mailReturnFailure!</td>
</tr>
<tr>
<td>mailReturnInsufficientMemory!</td>
</tr>
<tr>
<td>mailReturnInvalidMessage!</td>
</tr>
<tr>
<td>mailReturnUserAbort!</td>
</tr>
<tr>
<td>mailReturnDiskFull!</td>
</tr>
</tbody>
</table>

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:

```powerScript
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."
) RETURN
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:

```
mailRecipient recip
mailMessage msg
mailReturnCode mRet
string s_ID

mRet = mSes.mailGetMessages()
IF mRet <> mailReturnSuccess! THEN
    MessageBox("No Messages", "Inbox empty.")
    RETURN
END IF
s_ID = mSes.MessageID[UpperBound(mSes.MessageID)]
mRet = mSes.mailReadMessage(s, msg, &
    mailEntireMessage!, FALSE )
IF mRet <> mailReturnSuccess! THEN
    MessageBox("Message", "Can't read message.")
    RETURN
END IF

msg.Subject = msg.Subject + " Test"
recip.Name = "Smith, Jane"
mRet = mSes.mailResolveRecipient( recip )
msg.Recipient[1] = recip
mRet = mSes.mailSaveMessage(s_ID, msg)
IF mRet <> mailReturnSuccess! THEN
    MessageBox("Save Old Message", "Failed somehow.")
END IF
```

See also the mail examples in the samples that are supplied with PowerBuilder.

See also

- mailReadMessage
- mailResolveRecipient
mailSend

Description
Sends a mail message. If no message information is supplied, the mail system provides a dialog box for entering it before sending the message.

Applies to
mailSession object

Syntax
\texttt{mailSession.mailSend ( \{ \texttt{mailMessage} \})}

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 \texttt{null}, mailSend returns \texttt{null}.

Usage
Before calling mail functions, you must declare and create a mailSession object and call \texttt{mailLogon} to establish a mail session.

For mailSend, mailOriginator! is not a valid value for the Recipient property of the \texttt{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:

\begin{verbatim}
mailSession mSes
mailReturnCode mRet
mailMessage mMsg

// Create a mail session
mSes = create mailSession
\end{verbatim}
// 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><code>string</code></td>
<td>The string in which you want to look for a pattern of characters</td>
</tr>
<tr>
<td><code>textpattern</code></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.
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;) 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 ABBBC, 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-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>
Examples

This statement returns true if the text in sle_ID begins with one or more uppercase or lowercase letters (^ at the beginning of the pattern means that the beginning of the string must match the characters that follow):

\[ \text{Match(sle_ID.Text, } ^{\text{\textasciitilde}}[A-Za-z]\text{)} \]

This statement returns false if the text in sle_ID contains any digits (^ inside a bracket is a complement operator):

\[ \text{Match(sle_ID.Text, } [^0-9]\text{)} \]

This statement returns true if the text in sle_ID contains one uppercase letter:

\[ \text{Match(sle_ID.Text, } [A-Z]\text{)} \]

This statement returns true if the text in sle_ID contains one or more uppercase letters (+ indicates one or more occurrences of the pattern):

\[ \text{Match(sle_ID.Text, } [A-Z]+\text{)} \]

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):

\[ \text{Match(sle_ID.Text, } ^{\text{\textasciitilde}}[0-9][0-9][A-Za-z]\text{)} \]

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**

```
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, Max returns null.

**Usage**
If either of the values being compared is null, Max returns null.

**Examples**
This statement returns 7:

```
Max (4, 7)
```

This statement returns -4:

```
Max (-4, -7)
```

This statement returns 8.2, a decimal value:

```
Max (8.2, 4)
```

**See also**
Min

Max method for DataWindows in the 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**

```
olestorage.MemberDelete ( membername )
```

<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 member (substorage or stream) you want to delete</td>
</tr>
<tr>
<td>membername</td>
<td>A string specifying the name of the member you want to delete from 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 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:

```powerScript
boolean lb_memexists
integer result

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

stg_stuff.MemberExists("wordobj", lb_memexists)
IF lb_memexists THEN
    result = stg_stuff.MemberDelete("wordobj")
    IF result = 0 THEN stg_stuff.Save()
END IF
```

**See also**
MemberExists
MemberRename
Open
**MemberExists**

**Description**
Determines whether the named member is part of an OLE object in a storage. The member can be another OLE object (a substorage) or a stream.

**Applies to**
OLEStorage objects

**Syntax**
`olestorage.MemberExists( membername, exists )`

<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 that you want to check</td>
</tr>
<tr>
<td><code>membername</code></td>
<td>A string whose value is the name of the member that you want to check</td>
</tr>
<tr>
<td><code>exists</code></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:
- -1 The storage is not open
- -9 Other error

If any argument’s value is null, `MemberExists` returns null.

**Examples**
This example creates a storage object and opens an OLE object in a file. It checks whether `wordobj` is a substorage within that object and, if so, deletes it and saves the object back to the file:

```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**
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

\texttt{olestorage.MemberRename ( membername, newname )}

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 \textit{wordobj} is a substorage within that object, and if so renames it to memo and saves the object back to the file:

\begin{verbatim}
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()
ENDIF
\end{verbatim}

See also

MemberDelete
MemberExists
Open
**MessageBox**

**Description**
Displays a system MessageBox with the title, text, icon, and buttons you specify.

**Syntax**
```
MessageBox ( title, text {, icon {, button {, default } } } )
```

**Argument** | **Description**
---|---
`title` | A string specifying the title of the message box, which appears in the box’s title bar.
`text` | The text you want to display in the message box. The text can be a numeric datatype, a string, or a boolean value.
`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):

```
MessageBox(“Greeting”, “Hello User”) 
```

The following statements display a MessageBox titled Result and containing the result of a function, the Exclamation icon, and the OK and Cancel buttons (the Cancel button is the default):

```
integer Net
long Distance = 3.457

Net = MessageBox("Result", Abs(Distance), &
    Exclamation!, OKCancel!, 2)
IF Net = 1 THEN
    ... // Process OK.
ELSE
    ... // Process CANCEL.
END IF
```
Mid

Description
Obtains a specified number of characters from a specified position in a string.

Syntax
Mid( string, start [, length ] )

<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.</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:

```powershell
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:

```powerScript
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:

```powerScript
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`.
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 (optional)</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.

See also
AscA
Left
Pos
Right
UpperBound
Mid method for DataWindows in the DataWindow Reference or the online Help
**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**
Min (x, y)

**Argument** | **Description**
---|---
x | The number to which you want to compare y
y | The number to which you want to compare x

**Return value**
The datatype of x or y, whichever datatype is more precise. If any argument’s value is null, Min returns null.

**Usage**
If either of the values being compared is null, Min returns null.

**Examples**
This statement returns 4:
Min (4, 7)

This statement returns -7:
Min (-4, -7)

This statement returns 3.0, a decimal value:
Min (9.2, 3.0)

**See also**
Max
Min 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**

```plaintext
Minute( 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 minutes</td>
</tr>
</tbody>
</table>

**Return value**
Integer. Returns the minutes portion of `time` (00 to 59). If `time` is null, `Minute` returns null.

**Examples**
This statement returns 1:

```plaintext
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**

```plaintext
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:

```plaintext
Mod(20, 6)
```

This statement returns 1.5:

```plaintext
Mod(25.5, 4)
```

This statement returns 2.5:

```plaintext
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).

**Syntax**
```
colorname.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:

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

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).

Syntax
controlname.ModifyData ( seriesnumber, datapoint, xvalue, yvalue )

Argument | Description
---|---
controlname | The name of the scatter graph in which you want to modify data in a series
seriesnumber | The number that identifies the series in which you want to modify data
datapoint | The number of the data point for which you want to modify data
xvalue | The new x value of the data you want to modify
yvalue | The new y value of the data you want to modify

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, ModifyData returns null.

Usage
For scatter graphs, there are no categories. You specify the position in the series whose data you want to modify and provide the x and y values for the data.

Examples
These statements modify the data point 9 in the series named Test One in the scatter graph gr_product_data:

    integer SeriesNbr
    SeriesNbr = gr_product.FindSeries("Test One")
gr_product_data.ModifyData(SeriesNbr, &
    9, 4.55, 86.38)

See also
AddData
FindSeries

732
**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)
```

These statements store in `start_month` the month entered in the SingleLineEdit `sle_start_date`:

```
integer start_month
start_month = Month(date(sle_start_date.Text))
```

See also
Day
Date
Year
Month 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

```
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:

```powerbuilder
objectname.X = x
objectname.Y = y
```

These statements cause PowerBuilder to redraw gb_box1 twice:

```powerbuilder
gb_box1.X = 150
gb_box1.Y = 200
```

This statement has the same result but redraws gb_box1 once:

```powerbuilder
gb_box1.Move(150,200)
```

Examples

This statement changes the X and Y properties of gb_box1 to 150 and 200, respectively, and moves gb_box1 to the new location:

```powerbuilder
gb_box1.Move(150, 200)
```

This statement moves the picture p_Train2 next to the picture p_Train1:

```powerbuilder
```
MoveTab

Description
Moves a tab page to another position in a Tab control, changing its index number.

Applies to
Tab controls

Syntax

\[
\text{tabcontrolname}.\text{MoveTab}(\text{source, destination})
\]

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:

\[
\text{tab}_1.\text{MoveTab}(1, 0)
\]

This example move the fourth tab to the first position:

\[
\text{tab}_1.\text{MoveTab}(4, 1)
\]

This example move the fourth tab to the third position:

\[
\text{tab}_1.\text{MoveTab}(4, 3)
\]

See also
OpenTab
SelectTab
_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)`

Return value
Long. Returns 0 if it succeeds and a negative number if an error occurs.

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:

```pascal
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:

```pascal
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()

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:

```power_script
String ls_filename, ls_line
TraceFile ltf_file
TraceActivityNode ltan_node

ls_filename = sle_filename.text
ltf_file = CREATE TraceFile
ltf_file.Open(ls_filename)

ls_line = "CollectionTime = " + &
    String(ltf_file.CollectionTime) + "~r~n" + &
    "Num Activities = " + &
    String(ltf_file.NumberOfActivities) + "~r~n"
mle_output.text = ls_line

ltan_node = ltf_file.NextActivity()
```
**Now**

Obtains the current time based on the system time of the client machine.

**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
CHAPTER 10 PowerScript Functions

ObjectAtPointer

Description
Finds out where the user clicked in a graph. ObjectAtPointer reports the region of the graph under the pointer and stores the associated series and data point numbers in the designated variables.

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

Syntax
controlname.ObjectAtPointer ( { graphcontrol, } seriesnumber, datapoint )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the graph object for which you want the object under the pointer, or 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 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. If any argument’s value is null, ObjectAtPointer also returns null.

Values of grObjectType and the parts of the graph associated with them are:
- TypeCategory! – A label for a category
- TypeCategoryAxis! – The category axis or between the category labels
- TypeCategoryLabel! – The label of the category axis
- TypeData! – A data point or other data marker
- TypeGraph! – Any place within the graph control that 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
ObjectAtPointer

- 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, ObjectAtPointer stores the data point number in the designated variable.

When the user clicks in a series, but not on the actual data point, ObjectAtPointer stores 0 in datapoint and when the user clicks in a category, ObjectAtPointer stores 0 in seriesnumber. When the user clicks other parts of the graph, ObjectAtPointer stores 0 in both variables.

Call ObjectAtPointer first

ObjectAtPointer is most effective as the first function call in the script for the Clicked event for the graph control. Make sure you enable the graph control (the default is disabled). Otherwise, the Clicked event script is never run.

Examples

These statements store the series number and data point number at the pointer location in the graph named gr_product in SeriesNbr and ItemNbr. If the object type is TypeSeries! they obtain the series name, and if it is TypeData! they get the data value:

```powerbuilder
integer SeriesNbr, ItemNbr
double data_value
grObjectType object_type
string SeriesName

object_type = &
gr_product.ObjectAtPointer(SeriesNbr, ItemNbr)

IF object_type = TypeSeries! THEN
    SeriesName = &
gr_product.SeriesName(SeriesNbr)
ELSEIF object_type = TypeData! THEN
    data_value = &
gr_product.GetData(SeriesNbr, ItemNbr)
END IF
```
These statements store the series number and data point number at the pointer location in the graph named `gr_computers` in the DataWindow control `dw_equipment` in `SeriesNbr` and `ItemNbr`:

```plaintext
integer SeriesNbr, ItemNbr
dw_equipment.ObjectAtPointer("gr_computers", & SeriesNbr, ItemNbr)
```

See also
AddData
AddSeries

---

### 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**

```plaintext
jaguarorb.Object_To_String ( 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>object</code></td>
<td>The CORBA object that will be converted to a string. 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.</td>
</tr>
</tbody>
</table>

**Return value**

String. Returns the string representation of a CORBA object.

**Usage**

The `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.

`Object_To_String` is typically used in conjunction with `String_To_Object`, which allows you to deserialize an object reference.
The following example shows the use of the `Object_To_String` function to serialize a proxy object reference:

```
Payroll payroll
JaguarORB my_orb
...
my_orb = CREATE JaguarORB
my_orb.init("ORBRetryCount=3,ORBRetryDelay=1000")
...
String payroll_ior = my_orb.Object_To_String(payroll)
```

**See also**
- Init
- `String_To_Object`

---

### 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.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</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>

**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>
<tr>
<td>An OLE object in an OLE storage file and load it into a storage object in memory</td>
<td>Syntax 5</td>
</tr>
<tr>
<td>An OLE object that is a member of an open OLE storage and load it into a storage object in memory</td>
<td>Syntax 6</td>
</tr>
<tr>
<td>A stream in an OLE storage object in memory and load it into a stream object</td>
<td>Syntax 7</td>
</tr>
</tbody>
</table>

**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>

**Syntax 1**

**Description**

Opens a window object of a known datatype. **Open** displays the window and makes all its properties and controls available to scripts.

**Applies to**

Window objects
Open

Syntax

Open ( windowvar {, parent } )

<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 (child and pop-up windows only) (optional)</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.

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`:

```powerscript
Open(w_employee)
```

The following statements open an instance of a window of the type `w_employee`:

```powerscript
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:

```powerscript
child cw_data
Open(cw_data, w_employee)
```

The following code opens two windows of type `w_emp`:

```powerscript
w_emp w_el, w_e2
Open(w_el)
Open(w_e2)
```

See also

- `Close`
- `OpenWithParm`
- `Show`
Open

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
`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 \textit{windowtype} and places a reference to this instance in the variable \textit{windowvar}.

If \textit{windowtype} is a descendent window, you can only reference properties, events, functions, or structures that are part of the definition of \textit{windowvar}. For example, if a user event is declared for \textit{windowtype}, you cannot reference it.

The object specified in \textit{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 \textit{s\_w\_name} and stores the reference to the window in the variable \textit{w\_to\_open}. The \texttt{SELECT} statement retrieves data specifying the window type from the database and stores it in \textit{s\_w\_name}:

\begin{verbatim}
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)
\end{verbatim}

This example opens an array of ten windows of the type specified in the string \textit{is\_w\_empl} and assigns a title to each window in the array. The string \textit{is\_w\_empl} is an instance variable whose value is a window type:

\begin{verbatim}
integer n
window win_array[10]

FOR n = 1 to 10
  Open(win_array[n], is_wempl)
  win_array[n].title = "Window " + string(n)
NEXT
\end{verbatim}

The following statements open four windows. The type of each window is stored in the array \textit{w\_stock\_type}. The window reference from the \texttt{Open} function is assigned to elements in the array \textit{w\_stock\_win}:

\begin{verbatim}
window w_stock_win[]
string w_stock_type[4]
\end{verbatim}
Open

```
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 )
```

<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 into which you want to load an OLE object.</td>
</tr>
<tr>
<td><code>OLEsourcefile</code></td>
<td>A string specifying the name of an OLE storage file containing the object. The file must already exist and contain an OLE object. <code>OLEsourcefile</code> can include a path for the file, as well as path information inside the OLE 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 found or its data has an invalid format
- `-9` Other error

If any argument’s value is null, Open returns null.

**Examples**
This example opens the object in the file `MYSTUFF.OLE` and loads it into in the control `ole_1`:
```
integer result
result = ole_1.Open("c:\ole2\mystuff.ole")
```

See also
InsertFile
Save
SaveAs
## Syntax 4

**For opening an OLE object in memory into a control**

**Description**

Opens an OLE object that is in a OLE storage object in memory and loads it into an OLE control.

**Applies to**

OLE controls

**Syntax**

```powerscript
olecontrol.Open (sourcestorage, substoragename)
```

<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 into which you want to load an OLE object</td>
</tr>
<tr>
<td><code>sourcestorage</code></td>
<td>The name of an object variable of OLEStorage containing the object you want to load into <code>olecontrol</code></td>
</tr>
<tr>
<td><code>substoragename</code></td>
<td>A string specifying the name of a substorage that contains the desired object within <code>storagename</code></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
-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:

```powerscript
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:

```powerscript
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**

```plaintext
olestorage.Open ( OLEsourcefile {, readmode {, sharemode } } )
```

<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 into which you want to load the OLE object.</td>
</tr>
<tr>
<td><code>OLEsourcefile</code></td>
<td>A string specifying the name of an OLE storage file containing the object. The file must already exist and contain OLE objects. <code>OLEsourcefile</code> can include the file’s path, as well as path information within the storage.</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>OLEsourcefile</code>. Values are:</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>OLEsourcefile</code> will fare. Values are:</td>
</tr>
</tbody>
</table>

- **stgReadWrite** – (Default) Read/Write access. If the file does not exist, `Open` creates it.
- **stgRead** – Read-only access. You cannot change `OLEsourcefile`.
- **stgWrite** – Write access. You can rewrite `OLEsourcefile` but not read its current contents. If the file does not exist, `Open` creates it.
- **stgExclusive** – (Default) No other attempt to open `OLEsourcefile` will succeed.
- **stgDenyNone** – Any other attempt to open `OLEsourcefile` will succeed.
- **stgDenyRead** – Other attempts to open `OLEsourcefile` for reading will fail.
- **stgDenyWrite** – Other attempts to open `OLEsourcefile` for writing will fail.
CHAPTER 10  PowerScript Functions

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 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:

```powerbuilder
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:

```powerbuilder
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.

```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
```

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.

```powerbuilder
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**

```powerscript
olestorage.Open ( substoragename, readmode, sharemode, sourcestorage )
```

<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 into which you want to load the OLE object.</td>
</tr>
<tr>
<td><code>substoragename</code></td>
<td>A string specifying the name of the storage member within <code>sourcestorage</code> that you want to open. Note the reversed order of the <code>sourcestorage</code> and <code>substoragename</code> arguments from Syntax 4.</td>
</tr>
<tr>
<td><code>readmode</code></td>
<td>A value of the enumerated datatype <code>stgReadMode</code> that specifies the type of access you want for <code>substoragename</code>. Values are:</td>
</tr>
<tr>
<td></td>
<td>- <code>stgReadWrite!</code> – Read/write access. If the member does not exist, <code>Open</code> creates it.</td>
</tr>
<tr>
<td></td>
<td>- <code>stgRead!</code> – Read-only access. You cannot change <code>substoragename</code>.</td>
</tr>
<tr>
<td></td>
<td>- <code>stgWrite!</code> – Write access. You can rewrite <code>substoragename</code> but not read its current contents. If the member does not exist, <code>Open</code> creates it.</td>
</tr>
<tr>
<td><code>sharemode</code></td>
<td>A value of the enumerated datatype <code>stgShareMode</code> that specifies how other attempts, by your own or other applications, to open <code>substoragename</code> will fare. Values are:</td>
</tr>
<tr>
<td></td>
<td>- <code>stgExclusive!</code> – (Default) No other attempt to open <code>substoragename</code> will succeed.</td>
</tr>
<tr>
<td></td>
<td>- <code>stgDenyNone!</code> – Any other attempt to open <code>substoragename</code> will succeed.</td>
</tr>
<tr>
<td></td>
<td>- <code>stgDenyRead!</code> – Other attempts to open <code>substoragename</code> for reading will fail.</td>
</tr>
<tr>
<td></td>
<td>- <code>stgDenyWrite</code> – Other attempts to open <code>substoragename</code> for writing will fail.</td>
</tr>
<tr>
<td><code>sourcestorage</code></td>
<td>An open OLEStorage object containing <code>substoragename</code>.</td>
</tr>
</tbody>
</table>

**Return value**

Return value

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

-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.

```pascal
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

754 PowerBuilder
**Syntax 7**

**For opening OLE streams**

Opens a stream in an open OLE storage object and loads it into an OLE stream object.

**Applies to**

OLE stream objects

**Syntax**

`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>
Open

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.

```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
Syntax 8  For opening trace files

Description
Opens the specified trace file for reading.

Applies to
TraceFile object

Syntax
instancename.Open (filename)

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:

```PowerScript
TraceFile ltf_file
String ls_filename

ltf_file = CREATE TraceFile
ltf_file.Open(ls_filename)
...
```

See also
Close
Reset
NextActivity
OpenChannel

**Description**
Opens a channel to a DDE server application.

**Syntax**

```
OpenChannel ( applname, topicname {, windowhandle } )
```

**Argument** | **Description**
--- | ---
applname | A string specifying the DDE name of the DDE server application.

topicname | A string identifying the data or the instance of the application you want to use (for example, in Microsoft Excel, the topic name could be System or the name of an open spreadsheet).

windowhandle (optional) | The handle of the window that you want to act as the DDE client. Specify this parameter to control which window is acting as the DDE client when you have more than one open window.

**Return value**
Long. Returns the handle to the channel (a positive integer) if it succeeds. If an error occurs, OpenChannel returns a negative integer. Values are:

-1 | Open failed
-9 | Handle is null

**Usage**
Use OpenChannel to open a channel to a DDE server application and leave it open so you can efficiently execute more than one DDE request. This type of DDE conversation is called a warm link. Because you open a channel, the operating system does not have to poll all open applications every time you send or ask for data.

The following is an outline of a warm-link conversation:

- Open a DDE channel with OpenChannel and check that it returns a valid channel handle (a positive value).
- Execute several DDE functions. You can use the following functions:
  
  ```
  ExecRemote ( command, handle, <windowhandle> )
  GetRemote ( location, target, handle, <windowhandle> )
  SetRemote ( location, value, handle, <windowhandle> )
  ```

- Close the DDE channel with CloseChannel.

If you only need to use a remote DDE function once, you can call ExecRemote, GetRemote, or SetRemote without opening a channel. This is called a cold link. Without an open channel, the operating system polls all running applications to find the specified server application each time you call a DDE function.

Your PowerBuilder application can also be a DDE server.
For 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:

```powerbuilder
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:

```powerbuilder
long handle
string s_regiondata[3]
handle = OpenChannel("Excel", "REGION.XLS", &
Handle(w_ddewin))
GetRemote("R1C2", s_regiondata[1], handle, &
Handle(w_ddewin))
GetRemote("R1C3", s_regiondata[2], handle, &
Handle(w_ddewin))
GetRemote("R1C4", s_regiondata[3], handle, &
Handle(w_ddewin))
CloseChannel(handle, Handle(w_ddewin))
```

**See also**
CloseChannel
ExecRemote
GetRemote
SetRemote
**OpenSheet**

**Description**
Opens a sheet within an MDI (multiple document interface) frame window and creates a menu item for selecting the sheet on the specified menu.

**Applies to**
Window objects

**Syntax**
```
OpenSheet ( sheetrefvar {, windowtype }, mdiframe {, position {, arrangeopen } } )
```

**Argument** | **Description**
---|---
`sheetrefvar` | The name of any window variable that is not an MDI frame window. `OpenSheet` places a reference to the open sheet in `sheetrefvar`.
`windowtype` (optional) | A string whose value is the 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` (optional) | The number of the menu item (in the menu associated with the sheet) to which you want to append the names of the open sheets. Menu bar menu items are numbered from the left, beginning with 1. The default value of 0 lists the open sheets under the next-to-last menu item.
`arrangeopen` (optional) | A value of the ArrangeOpen enumerated 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 \textit{windowtype}, the window object specified in \textit{windowtype} must be the same datatype as \textit{sheetrefvar} (a datatype includes datatypes inherited from it). The datatype of \textit{sheetrefvar} is usually window, from which all windows are inherited, but it can be any ancestor of \textit{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 \textit{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 \textit{PowerBuilder User’s Guide}.

\textit{OpenSheet} opens a sheet and appends its name to the item on the menu bar specified in \textit{position}. If \textit{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 \textit{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 \textit{position} and a final item \textit{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.

\textbf{Opening response windows}

\textit{Do not} use the \textit{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:

```powershell
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
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:

```powershell
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.
CHAPTER 10  PowerScript Functions

Applies to

Window objects

Syntax

**OpenSheetWithParm** ( `sheetrefvar`, `parameter {}`, `windowtype {, position {, arrangeopen} }` )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sheetrefvar</code></td>
<td>The name of any window variable that is not an MDI frame window. <strong>OpenSheetWithParm</strong> places a reference to the open sheet in <code>sheetrefvar</code>.</td>
</tr>
<tr>
<td><code>parameter</code></td>
<td>The parameter you want to store in the Message object when the sheet is opened. <strong>Parameter</strong> 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>sheetrefvar</code>.</td>
</tr>
<tr>
<td>(optional)</td>
<td><strong>mdiframe</strong> The name of the MDI frame window in which you want to open this sheet.</td>
</tr>
<tr>
<td><code>position</code></td>
<td>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>(optional)</td>
<td><strong>arrangeopen</strong> 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>(optional)</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.

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:

```pascal
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 \texttt{w\_child\_1} as an MDI sheet. The name of the sheet is appended to the fourth menu item associated with \texttt{w\_child\_1} and is layered.

The script also passes a reference to the SingleLineEdit control \texttt{sle\_state} as a \texttt{PowerObject} parameter of the Message object. The script for the Open event of \texttt{w\_child\_1} uses the text in the edit control to determine what type of calculations to perform. Note that this would fail if \texttt{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 \texttt{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:

\begin{verbatim}
window child
    OpenSheetWithParm(child, sle_state, "w\_child\_1", MDI\_User, 4, Layered!)
\end{verbatim}

The second script, for the Open event in \texttt{w\_child\_1}, is:

\begin{verbatim}
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
\end{verbatim}

See also

\begin{itemize}
    \item ArrangeSheets
    \item OpenSheet
\end{itemize}
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</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>user object’s type at runtime</td>
<td></td>
</tr>
</tbody>
</table>

Syntax 1
For user objects of a known datatype

Opens a custom visual user object of a known datatype as a tab page in a Tab control.

Apply to
Tab controls

Syntax

tabcontrolname.OpenTab ( userobjectvar, 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>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. OpenTab places a reference to the opened custom visual user object in 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.

Opening the same object twice
If you call Syntax 1 twice to open the same user object, PowerBuilder does open the user object again as another tab page, in contrast to the behavior of Open and OpenUserObject.

Examples
This statement opens an instance of a user object named u_Employee as a tab page in the Tab control tab_1:

```
    tab_1.OpenTab(u_Employee, 0)
```

The following statements open an instance of a user object u_to_open as a tab page in the Tab control tab_1. It becomes the first tab in the control:

```
    u_employee u_to_open
    tab_1.OpenTab(u_to_open, 1)
```

See also
OpenTabWithParm

Syntex 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.

Applies to
Tab controls

Syntax
```
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>
</tbody>
</table>
OpenTab

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.

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 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 opens a user object as the last tab page in the Tab control tab_1. The user object is of the type specified in the string s_u_name and stores the reference to the user object in the variable u_to_open:

```
UserObject u_to_open
string s_u_name

s_u_name = sle_user.Text
tab_1.OpenTab(u_to_open, s_u_name, 0)
```

See also

OpenTabWithParm

OpenTabWithParm

Adds a visual user object to the specified window and makes all its properties and controls available to scripts, as OpenTab does. OpenTabWithParm also stores a parameter in the system’s Message object so that it is accessible to the opened object.

<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 and stores a parameter in the system’s Message object.

Applies to

Tab controls

Syntax

```
tabcontrolname.OpenTabWithParm ( userobjectvar, parameter, 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>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.</td>
</tr>
</tbody>
</table>
OpenTabWithParm

**Argument**

- **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.

<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.

See also the usage notes for OpenTab, all of which apply to OpenTabWithParm.
Examples

This statement opens an instance of a user object named \texttt{u\_Employee} as a tab page in the Tab control \texttt{tab\_empsettings}. It also stores the string \texttt{James Newton} in \texttt{Message.StringParm}. The Constructor event script for the user object uses the string parameter as the text of a StaticText control \texttt{st\_empname} in the object. The script that opens the tab page has the following statement:

\begin{verbatim}
    tab_empsettings.OpenTabWithParm(u_Employee, "James Newton", 0)
\end{verbatim}

The user object's Constructor event script has the following statement:

\begin{verbatim}
    st_empname.Text = Message.StringParm
\end{verbatim}

The following statements open an instance of a user object \texttt{u\_to\_open} as the first tab page in the Tab control \texttt{tab\_empsettings} and store a number in \texttt{message.DoubleParm}. The last statement selects the tab page:

\begin{verbatim}
    u_employee u_to_open
    integer age = 50
    tab_1.OpenTabWithParm(u_to_open, age, 1)
    tab_1.SelectTab(u_to_open)
\end{verbatim}

See also

OpenTab

\section*{Syntax 2}

\textbf{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, \texttt{OpenTabWithParm} stores a parameter in the system's \texttt{Message} object so that it is accessible to the opened object.

\textbf{Description}

\texttt{OpenTabWithParm} places a reference to the opened user object in \texttt{userobjectvar}.

\textbf{Applies to}

Tab controls

\textbf{Syntax}

\begin{verbatim}
    tabcontrolname.OpenTabWithParm( userobjectvar, parameter, 
        userobjecttype, index )
\end{verbatim}

\begin{tabular}{|l|l|}
\hline
\textbf{Argument} & \textbf{Description} \\
\hline
\texttt{tabcontrolname} & The name of the Tab control in which you want to open the user object as a tab page. \\
\hline
\texttt{userobjectvar} & A variable of datatype UserObject. \texttt{OpenTabWithParm} places a reference to the opened user object in \texttt{userobjectvar}. \\
\hline
\end{tabular}
OpenTabWithParm

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>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, 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.

See also the usage notes for OpenTab, all of which apply to OpenTabWithParm.
Examples

The following statement opens an instance of a user object \texttt{u\_data} of type \texttt{u\_benefit\_plan} as the last tab page in the Tab control \texttt{tab\_1}. The parameter "Benefits" is stored in message.StringParm:

\begin{verbatim}
UserObject u\_data
\hspace{1em}tab\_1.OpenTabWithParm(u\_data, \& "Benefits", "u\_benefit\_plan", 0)
\end{verbatim}

These statements open a user object of the type specified in the string \texttt{s\_u\_name} and store the reference to the user object in the variable \texttt{u\_to\_open}.

The script gets the value of \texttt{s\_u\_name}, the type of user object to open, from the database. The parameter is the text of the SingleLineEdit \texttt{sle\_loc}, so it is stored in Message.StringParm. The user object becomes the third tab page in the Tab control \texttt{tab\_1}:

\begin{verbatim}
UserObject u\_to\_open
\hspace{1em}string s\_u\_name, e\_location
\hspace{1em}e\_location = sle\_location.Text
\hspace{1em}SELECT next_userobj INTO : s\_u\_name FROM routing\_table WHERE ... ;
\hspace{1em}tab\_1.OpenTabWithParm(u\_to\_open, \& e\_location, s\_u\_name, 3)
\end{verbatim}

The following statements open a user object of the type specified in the string \texttt{s\_u\_name} and store the reference to the user object in the variable \texttt{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 \texttt{tab\_1}:

\begin{verbatim}
UserObject u\_to\_open
\hspace{1em}integer age = 60
\hspace{1em}string s\_u\_name
\hspace{1em}s\_u\_name = sle\_user.Text
\hspace{1em}tab\_1.OpenTabWithParm(u\_to\_open, age, \& s\_u\_name, 1)
\end{verbatim}

See also

\begin{verbatim}
OpenTab
\end{verbatim}
**OpenUserObject**

Adds 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**

Opens a user object of a known datatype.

**Description**

Applies to Window objects

**Syntax**

```powerbuilder
windowname.OpenUserObject ( userobjectvar {, x, y} )
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windowname</code></td>
<td>The name of the window in which you want to open the user object.</td>
</tr>
<tr>
<td><code>userobjectvar</code></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. <code>OpenUserObject</code> places a reference to the opened user object in <code>userobjectvar</code>.</td>
</tr>
<tr>
<td><code>x</code> (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><code>y</code> (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 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.

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):

```powerscript
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:

```powerscript
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:

```powerscript
w_info.OpenUserObject(u_data, 20, 100)
```

See also
OpenUserObjectWithParm

Syntax 2

For user objects of unknown datatype
Opens a user object when the datatype of the user object is not known until the script is executed.

Applies to
Window objects

Syntax
```powerscript
windowname.OpenUserObject( userobjectvar, userobjecttype {, x, y} )
```
OpenUserObject

Argument | Description
---|---
`windowname` | The name of the window in which you want to open the user object.
`userobjectvar` | A variable of datatype DragObject. OpenUserObject places a reference to the opened user object in `userobjectvar`.
`userobjecttype` | A string whose value is the name of the user object you want to display. The datatype of `userobjecttype` must be a descendant of `userobjectvar`.
`x` (optional) | The x coordinate in PowerBuilder units of the user object within the window’s frame. The default is 0.
`y` (optional) | The y coordinate in PowerBuilder units of the user object within the window’s frame. The default is 0.

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, OpenUserObject returns null.

Usage

Use Syntax 1 when you know what user object you want to open. Use Syntax 2 when the application will determine what type of user object to open when the script runs.

You must open a user object before you can access the properties of the user object. If you access the user object’s properties before you open it, an execution error will occur.

A user object that is part of a window’s definition (that is, it was added to the window in the Window painter) does not have to 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.

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:

```powershell
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>
OpenUserObjectWithParm

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 } )`

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. `OpenUserObjectWithParm` places a reference to the opened user object in `userobjectvar`.
`parameter` | The parameter you want to store in the Message object when the user object is opened. `Parameter` must have one of these datatypes:
- String
- Numeric
- PowerObject

`x` (optional) | The x coordinate in PowerBuilder units of the user object within the window’s frame. The default is 0.
`y` (optional) | The y coordinate in PowerBuilder units of the user object within the window’s frame. The default is 0.

Return value
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 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
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

```
windowname.OpenUserObjectWithParm( userobjectvar, parameter, userobjecttype, x, y )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windowname</code></td>
<td>The name of the window in which you want to open the user object.</td>
</tr>
<tr>
<td><code>userobjectvar</code></td>
<td>A variable of datatype DragObject. OpenUserObjectWithParm places a reference to the opened user object in <code>userobjectvar</code>.</td>
</tr>
<tr>
<td><code>parameter</code></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><code>userobjecttype</code></td>
<td>A string whose value is the datatype of the user object you want to open. The datatype of <code>userobjecttype</code> must be a descendant of <code>userobjectvar</code>.</td>
</tr>
<tr>
<td><code>x</code> (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><code>y</code> (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 \texttt{u\_data} of type \texttt{u\_benefit\_plan} at location 20,100 in the window \texttt{w\_hresource}. The parameter "Benefits" is stored in message.StringParm:

\begin{verbatim}
DragObject u_data
w_hresource.OpenUserObjectWithParm(u_data, &
  "Benefits", "u\_benefit\_plan", 20, 100)
\end{verbatim}

These statements open a user object of the type specified in the string \texttt{s\_u\_name} and store the reference to the user object in the variable \texttt{u\_to\_open}. The script gets the value of \texttt{s\_u\_name}, the type of user object to open, from the database. The parameter is the text of the SingleLineEdit \texttt{sle\_loc}, so it is stored in Message.StringParm. The user object is at the default coordinates 0,0 in the window \texttt{w\_info}:

\begin{verbatim}
DragObject u_to_open
string s_u_name, e_location

e_location = sle_location.Text

SELECT next_userobj INTO : s_u_name
FROM routing_table
WHERE ... ;

w_info.OpenUserObjectWithParm(u_to_open, &
  e_location, s_u_name)
\end{verbatim}
**OpenWithParm**

The following statements display a user object of the type specified in the string `s_u_name` and store the reference to the user object in the variable `u_to_open`. The parameter is numeric so it is stored in `message.DoubleParm`. The user object is at the coordinates 100,200 in the window `w_emp`:

```pascal
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</th>
<th>Use</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>
CHAPTER 10  PowerScript Functions

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

```
OpenWithParm (windowvar, parameter {, parent })
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windowvar</code></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. 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><code>parent</code></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 <code>parent</code>, PowerBuilder associates the window being opened with the currently active window.</td>
</tr>
<tr>
<td>(child and pop-up windows only)</td>
<td></td>
</tr>
<tr>
<td>(optional)</td>
<td></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.
**OpenWithParm**

**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
Syntax 2

For windows of unknown datatype

Description

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>windowvar</td>
<td>A window variable, usually of datatype window. OpenWithParm places a reference to the open window in windowvar.</td>
</tr>
<tr>
<td>parameter</td>
<td>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</td>
</tr>
<tr>
<td>windowtype</td>
<td>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 windowvar.</td>
</tr>
<tr>
<td>parent</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 parent, PowerBuilder associates the window being opened with the currently active window. (child and pop-up windows 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, 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 `s_w_name` and store the reference to the window in the variable `w_to_open`. The script gets the value of `s_w_name`, the type of window to open, from the database. The parameter in `e_location` is text, so it is stored in Message.StringParm:

```powershell
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 
\textit{c\_w\_name}, store the reference to the window in the variable \textit{wc\_to\_open}, and 
make \textit{w\_emp} the parent window of \textit{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"

    \texttt{OpenWithParm(wc_to_open, age, c_w_name, w_emp)}
\end{verbatim}

See also

\texttt{CloseWithReturn}
\texttt{Open}

### OutgoingCallList

**Description**

Provides a list of the calls to other routines included in a performance analysis model.

**Applies to**

ProfileLine and ProfileRoutine objects

**Syntax**

\begin{verbatim}
    instancename.\texttt{OutgoingCallList ( list, aggregate )}
\end{verbatim}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{instancename}</td>
<td>Instance name of the ProfileLine or ProfileRoutine object.</td>
</tr>
<tr>
<td>\textit{list}</td>
<td>An unbounded array variable of datatype ProfileCall in which \texttt{OutgoingCallList} stores a ProfileCall object for each call to other routines from within this routine. This argument is passed by reference.</td>
</tr>
<tr>
<td>\textit{aggregate} (ProfileRoutine only)</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**

\texttt{ErrorReturn}. Returns one of the following values:

- **Success!** – The function succeeded
- **ModelError** – The model does not exist
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.

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:

```powerbuilder
Long ll_cnt
ProfileCall lproc_call[]

lpro_model.BuildModel()
lpro_model.RoutineList(iprort_list)

FOR ll_cnt = 1 TO UpperBound(iprort_list)

   iprort_list[ll_cnt].OutgoingCallList(lproc_call, & TRUE)

   ... 

NEXT
```

See also  

- `BuildModel`
- `IncomingCallList`
PageCount

Description
Returns the total number of pages in the document in a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
rtename.PageCount ( )

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 ( )

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.
ParentWindow

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

`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 User's Guide*.

### Examples

These statements return the parent of `child_1`. The parent is a window of the datatype `Win1`:

```powershell
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:

```powershell
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**

```powershell
controlname.Paste()
```

**Argument**

`controlname`

**Description**

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 `controlname` is a DataWindow, text is pasted into the edit control over the current row and column. If `controlname` is a DropDownListBox or DropDownListPictureListBox, the AllowEdit property must be `true`
**Paste**

**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 than 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:

```powershell
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:

```powershell
mle_Comment2.Paste()
```

If the clipboard contains an OLE object, this statement makes it the contents of the control `ole_1` and returns 0:

```powershell
ole_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

<table>
<thead>
<tr>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>olecontrol</td>
</tr>
</tbody>
</table>

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

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.
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:

```powershell
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**

```powershell
rtename.PasteRTF ( richtextstring, { band } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rtename</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>
</tbody>
</table>
| `band` (optional) | A value of the Band enumerated datatype specifying the band into which the rich text data is pasted. Values are:
  - `Detail`! – The data is pasted into the detail band
  - `Header`! – The data is pasted into the header band
  - `Footer`! – The data is pasted into the footer band
  The default is the band that contains the insertion point. |

**Return value**

Long. Returns -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`:

```powershell
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 ( )

Argument | Description
---|---
olecontrol | The name of the OLE control into which you want to paste the
object on the clipboard

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

- 1 User canceled without selecting a paste option
- 1 No data found
- 9 Other error

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:

```powerShell
integer li_result
li_result = ole_1.PasteSpecial()
```

See also
LinkTo
Paste
PasteLink
**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, \( \text{Pi} \) returns null.

**Usage**
Use \( \text{Pi} \) 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 \( \text{id\_Rad} \), an instance variable of type double:

```
Pi(1) * \text{id\_Rad}^2

Pi(\text{id\_Rad}^2)
```

The following statements compute the cosine of a 45-degree angle:

```
real degree = 45.0, cosine
cosine = \text{Cos}(\text{degree} * (\text{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>

---
**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:

```powershell
inget value
Value = PixelsToUnits(35, XPixelsToUnits!)
```

**See also**

UnitsToPixels

---

**Play**

**Description**

Starts playing an animation (an AVI clip).

**Applies to**

Animation controls

**Syntax**

`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.
This example starts playing an AVI clip at the first frame, plays to the last frame, and continues playing the clip indefinitely:

```powerbuilder
  integer li_return
  li_return = am_1.Play(0, -1, -1)
```

**See also**

Stop

---

**PointerX**

**Description**

Determines the distance of the pointer from the left edge of the specified object.

**Applies to**

Any object or control

**Syntax**

`objectname(PointerX)`

<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 left edge. If you do not specify <code>objectname</code>, <code>PointerX</code> reports the distance from the left edge of the current sheet or window.</td>
</tr>
</tbody>
</table>

**Return value**

Integer. Returns the pointer’s distance from the left edge of `objectname` in PowerBuilder units if it succeeds and -1 if an error occurs. If `objectname` is null, `PointerX` returns null.

**Examples**

In a script for a control in a window, the following example stores the distance of the pointer from the edge of the window in the variable `li_dist`. If the pointer is 5 units from the left edge of the window, `li_dist` equals 5:

```powerbuilder
  integer li_dist
  li_dist = Parent.PointerX()
```

This statement in a control’s RButtonDown script displays a pop-up menu `m_Appl.M_Help` at the cursor position:

```powerbuilder
  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:

```powerbuilder
  m_Appl.m_Help.PopMenu(This(PointerX()), &
                       This.PointerY())
```
### 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 )

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 allocate 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):

```powerscript
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:

```powerscript
NewMenu.m_language.PopMenu(&
    w_frame.PointerX(), w_frame.PointerY())
```

### PopulateError

**Description**
Fills in the Error object without causing a SystemError event.

**Syntax**

```
PopulateError( number, text )
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>The integer to be stored in the number property of the Error object</td>
</tr>
<tr>
<td>text</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:

```powerscript
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 (optional)</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:

```
long place_nbr
place_nbr = Pos(sle_group.Text, "NY")
sle_group.SelectText(place_nbr, 2)
sle_group.ReplaceText("North East")
```

These statements separate the return value of `GetBandAtPointer` into the band name and row number. The `Pos` function finds the position of the tab in the string and the `Left` and `Mid` functions extract the information to the left and right of the tab:

```
string s, ls_left, ls_right
integer li_tab

s = dw_groups.GetBandAtPointer()
li_tab = Pos(s, ",t", 1)
ls_left = Left(s, li_tab - 1)
ls_right = Mid(s, li_tab + 1)
```

You could write similar code for a generic parsing function with three arguments. The string `s` would be an argument passed by value and `ls_left` and `ls_right` would be strings passed by reference.

Other functions that return a pair of tab-separated values for which you could use the parsing function are `GetObjectAtPointer` and `GetValue`.

See also `GetValue` 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})

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>

PowerBuilder
CHAPTER 10  PowerScript Functions

**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()
```

**Argument** | **Description**
--- | ---
`editname` | 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

**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

Determines the line and column position of the insertion point or the start and end of selected text in an RichTextEdit control.

Applies to

RichTextEdit and DataWindow controls

Syntax

\textit{rtename}.\texttt{Position} ( \texttt{fromline}, \texttt{fromchar}, \texttt{tonline}, \texttt{tochar} )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{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>\texttt{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>\texttt{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>\texttt{tonline} (optional)</td>
<td>A long variable in which you want to save the number of the line where the selection ends.</td>
</tr>
<tr>
<td>\texttt{tochar} (optional)</td>
<td>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 \texttt{tonline} and \texttt{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 \texttt{fromchar} is set to 0. When the user drags down, the selection can end at the beginning of a line and \texttt{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 \texttt{tonline} and \texttt{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:

\[
\text{rte}_1.\text{Position}(ll\_line, ll\_char)
\]

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:

\[
\text{rte}_1.\text{Position}(ll\_startline, ll\_startchar, &
ll\_endline, ll\_endchar)
\]

- \( ll\_startline \) 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 \( st\_status \):

```powerscript
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 = cselect

ELSEIF l2 > l1 THEN // Selection, ins pt at end
  IF c2 = 0 THEN // End of selection (ins pt)
// at beginning of a line (char 0)
c2 = 999 // Change to end of prev line
l2 = l2 - 1
END IF

l1select = l1
c1select = c1
l2select = l2 + 1 // Add 1 to end line
c2select = c2

ELSEIF l2 < l1 THEN // selection, ins pt at start
  IF c1 = 0 THEN // End of selection (not ins pt)
    // at beginning of a line
    c1 = 999 // Change to end of prev line
    l1 = l1 - 1
  END IF
  l1select = l2
  c1select = c2
  l2select = l1 + 1 // Add 1 to end line
  // (start of selection)
c2select = c1
ELSE // l1 = l2, selection on one line
  l1select = l1
  l2select = l2 + 1 // Add 1 to line
  IF c1 < c2 THEN // ins pt at end
    c1select = c1
    c2select = c2
  ELSE // c1 > c2, ins pt at start
    c1select = c2
    c2select = c1
  END IF
END IF

// Select the extended selection
rtn = rte_1.SelectText( l1select, c1select, &
  l2select, c2select )

For an example of selecting each word in a RichTextEdit control, see
SelectTextWord.

See also
SelectedLine
SelectedStart
SelectText
Post

Description
Adds a message to the message queue for a window, either a PowerBuilder window or window of another application.

Syntax
Post (handle, message#, word, long)

Argument Description
<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 post a message.</td>
</tr>
<tr>
<td>message#</td>
<td>An UnsignedInteger whose value is the system message number of the message you want to post.</td>
</tr>
<tr>
<td>word</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
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

Adds an event to the end of the event queue of an object.

Applies to

Any object, except the application object

Syntax

`objectname.PostEvent ( event, { word, long } )`

<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> (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>

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:

```powerbuilder
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:

```powerscript
cb_OK.PostEvent(Clicked!)
```

This statement adds the user-defined event **cb_exit_request** to the event queue in the parent window:

```powerscript
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:

```powerscript
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:

```powerscript
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
\texttt{servicereference.PostURL (urlname, urldata, headers, \{serverport, \} data )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{servicereference}</td>
<td>Reference to the Internet service instance.</td>
</tr>
<tr>
<td>\texttt{urlname}</td>
<td>String specifying the URL to post.</td>
</tr>
<tr>
<td>\texttt{urldata}</td>
<td>Blob specifying arguments to the URL specified by \texttt{urlname}.</td>
</tr>
<tr>
<td>\texttt{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>\texttt{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>\texttt{data}</td>
<td>InternetResult instance into which the function returns HTML.</td>
</tr>
</tbody>
</table>

Return value
Integer. Returns values as follows:

1. Success
2. General error
3. 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.

\textit{Data} references a standard class user object that descends from InternetResult and that has an overridden \texttt{InternetData} function. This overridden function then performs the required processing with the returned HTML. Because the Internet returns data asynchronously, \texttt{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:

\texttt{Content-Type: application/x-www-form-urlencoded}

For more information on the InternetResult standard class user object and the \texttt{InternetData} function, use the PowerBuilder Browser.
Examples

This example calls the `PostURL` function using server port 8080. `inet` is an instance variable of type inet:

```power
Blob lblb_args
String ls_headers
String ls_url
Long ll_length

iir_msgbox = CREATE n_ir_msgbox
ls_url = "http://coltrane.sybase.com/"
ls_url += "cgi-bin/pbcgi60.exe/"
ls_url += "myapp/n_cst_html/f_test?"
lblb_args = blob(""
ll_length = Len(lblb_args)
ls_headers = "Content-Length: " &
            + String(ll_length) + "~n~n"
iinet.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:

```power
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
`rtnname, Preview (previewsetting)`

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

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} )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectname</td>
<td>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.</td>
</tr>
<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 of the left corner of the object, in thousandths of an inch.</td>
</tr>
<tr>
<td>y</td>
<td>An integer whose value is the y coordinate on the page of the left corner of the object, in thousandths of an inch.</td>
</tr>
<tr>
<td>width (optional)</td>
<td>An integer specifying the printed width of the object in thousandths of an inch. If omitted, PowerBuilder uses the object’s original width.</td>
</tr>
<tr>
<td>height (optional)</td>
<td>An integer specifying the printed height of the object in thousandths of an inch. If omitted, PowerBuilder uses the object’s original height.</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 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:

```powerbuilder
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:

```powerbuilder
long Job
Job = PrintOpen()
Print(Job, "Report of Year-to-Date Sales")
gr_sales1.Print(Job, 1000, PrintY(Job)+500, & 6000, 4500)
PrintPage(Job)
w_sales.Print(Job, 1000, 500, 6000, 4500)
PrintClose(Job)
```

**See also**

PrintCancel
PrintClose
PrintOpen
PrintScreen

**Syntax 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
Syntax

Print (printjobnumber, {tab1,} string {,tab2})

<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>tab1 (optional)</td>
<td>The position, measured from the left edge of the print area in thousandths of an inch, to which the print cursor should move before string is printed. If the print cursor is already at or beyond the position or if you omit tab1, Print starts printing at the current position of the print cursor.</td>
</tr>
<tr>
<td>string</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>tab2 (optional)</td>
<td>The new position, measured from the left edge of the print area in thousandths of an inch, of the print cursor after string printed. If the print cursor is already at or beyond the specified position, Print ignores tab2 and the print cursor remains at the end of the text. If you omit tab2, Print moves the print cursor to the beginning of a new line.</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") & \n+ " - " & \n+ String(Now(), "HH:MM:SS")
Print(li_prt, " ")
Print(li_prt, mle_message.text)

PrintClose(li_prt)
Syntax 3

For RichTextEdit controls

Prints the contents of a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax

\texttt{rtename.Print ( copies, pagerange, collate, canceldialog )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{rtename}</td>
<td>The name of the RichTextEdit control whose contents you want to print.</td>
</tr>
<tr>
<td>\texttt{copies}</td>
<td>An integer specifying the number of copies you want to print.</td>
</tr>
<tr>
<td>\texttt{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 \texttt{rtename} shares data with a DataWindow, \texttt{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>\texttt{collate}</td>
<td>A boolean value indicating whether you want the copies collated. Values are: TRUE – Collate copies FALSE – Do not collate copies</td>
</tr>
<tr>
<td>\texttt{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: TRUE – Display the dialog box 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:

\[
\text{rte}_1.\text{Print}(1, "1-5", \text{FALSE, TRUE})
\]

See also

Preview
PrintEx

---

**PrintBitmap**

**Description**

Writes a bitmap at the specified location on the current page.

**Syntax**

\[
\text{PrintBitmap}( \text{printjobnumber, bitmap, x, y, width, height} )
\]

**Argument** | **Description**
--- | ---
printjobnumber | The number the PrintOpen function assigned to the print job.
bitmap | A string whose value is the file name of the bitmap image.
x | An integer whose value is the x coordinate (in thousandths of an inch) on the page of the bitmap image.
y | An integer whose value is the y coordinate (in thousandths of an inch) on the page of the bitmap image.
width | The integer width of the bitmap image in thousandths of an inch. If width is 0, PowerBuilder uses the original width of the image.
height | The integer height of the bitmap image in thousandths of an inch. If height is 0, PowerBuilder uses the original height of the image.

**Return value**

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, PrintBitmap returns null.

**Usage**

PrintBitmap does not change the position of the print cursor, which remains where it was before the function was called. In general, print functions in which you specify coordinates do not affect the print cursor (see the functions listed in See also).

**Examples**

These statements define a new blank page and then print the bitmap in file d:\PB\BITMAP1.BMP in its original size at location 50,100:

```powerbuilder
long Job

// Define a new blank page.
Job = PrintOpen( )
```
// Print the bitmap in its original size.
PrintBitmap(Job, "d:\PB\BITMAP1.BMP", 50, 100, 0, 0)
// Send the page to the printer and close Job.
PrintClose(Job)

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 )

Argument  | Description
---|---
printjobnumber  | The number the PrintOpen function assigned to the print job

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If printjobnumber is null, PrintCancel returns null.

Usage
PrintCancel cancels the specified print job by deleting the spool file, if any, and closing the job. Because PrintCancel closes the print job, do not call the PrintClose function after you call PrintCancel.

Examples
In this example, a script for a Print button opens a print job and then opens a window with a cancel button. If the user clicks on the cancel button, its script sets a global variable that indicates that the user wants to cancel the job. After each printing command in the Print button’s script, the code checks the global variable and cancels the job if its value is true.

The definition of the global variable is:

boolean gb_printcancel
The script for the Print button is:

```powerScript
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:

```powerScript
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:

```pascal
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**
```powershell
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:

```powershell
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:</td>
</tr>
<tr>
<td></td>
<td>Default!</td>
</tr>
<tr>
<td></td>
<td>Fixed!</td>
</tr>
<tr>
<td></td>
<td>Variable!</td>
</tr>
<tr>
<td>fontfamily</td>
<td>A value of the FontFamily enumerated datatype indicating the family of the font:</td>
</tr>
<tr>
<td></td>
<td>AnyFont!</td>
</tr>
<tr>
<td></td>
<td>Decorative!</td>
</tr>
<tr>
<td></td>
<td>Modern!</td>
</tr>
<tr>
<td></td>
<td>Roman!</td>
</tr>
<tr>
<td></td>
<td>Script!</td>
</tr>
<tr>
<td></td>
<td>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:

```powerscript
long Job
Job = PrintOpen()
PrintDefineFont(Job, 1, "Courier 10Cpi", & -18, 400, Default!, Decorative!, FALSE, FALSE)
```

**See also**
`PrintClose`
`PrintOpen`
`PrintSetFont`

---

**PrintEx**
Prints the contents of a RichTextEdit control.

**Description**
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>
</tbody>
</table>
| `canceldialog`  | 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: TRUE – Display the dialog box
|                 | FALSE – Do not display the dialog box |
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 drivername ~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:

```
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
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

832 PowerBuilder
CHAPTER 10  PowerScript Functions

PrintOpen

Description
Opens a print job and assigns it a number, which you use in other printing statements.

Syntax
PrintOpen ( { jobname {, showprintdialog} } )

Return value
Long. Returns the job number if it succeds 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.
PrintOval

Examples
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>
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, 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:

```
long Job

// Define a new blank page.
Job = PrintOpen()

// Print an oval.
PrintOval(Job, 4000, 3000, 1000, 1000, 10)

... // Other printing
PrintClose(Job)
```

### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>An integer specifying the width in thousandths of an inch of the oval’s bounding box</td>
</tr>
<tr>
<td>height</td>
<td>An integer specifying the height in thousandths of an inch of the oval’s bounding box</td>
</tr>
<tr>
<td>thickness</td>
<td>An integer specifying the thickness of the line that outlines the oval in thousandths of an inch</td>
</tr>
</tbody>
</table>
PrintPage

See also PrintBitmap
      PrintClose
      PrintLine
      PrintOpen
      PrintRect
      PrintRoundRect

PrintPage

Description
Sends the current page to the printer or spooler and begins a new blank page in
the current print job.

Syntax
PrintPage ( printjobnumber )

Argument     Description
printjobnumber The number the PrintOpen function assigned to the print job

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s
value is null, PrintPage returns null.

Examples
This example opens a print job with a new blank page, prints a bitmap on the
page, and then sends the page to the printer and sets up a new blank page.
Finally, the last Print statement prints the company name on the new page:

    long Job

    // Open a job with new blank page.
    Job = PrintOpen()

    // Print a bitmap on the page.
    PrintBitmap(Job, "d:\PB\BITMAP1.BMP", 100,250, 0,0)

    // Begin a new page.
    PrintPage(Job)

    // Print the company name on the new page.
    Print(Job, "Sybase Corporation")

See also PrintClose
      PrintOpen

836  PowerBuilder
PrintRect

Description
Draws a white rectangle with a border of the specified thickness on the print page.

Syntax
PrintRect (printjobnumber, x, y, width, height, thickness)

Argument | Description
---|---
printjobnumber | The number the PrintOpen function assigned to the print job
x | An integer specifying the x coordinate in thousandths of an inch of the upper-left corner of the rectangle
y | An integer specifying the y coordinate in thousandths of an inch of the upper-left corner of the rectangle
width | An integer specifying the rectangle’s width in thousandths of an inch
height | An integer specifying the rectangle’s height in thousandths of an inch
thickness | An integer specifying the thickness of the rectangle’s border line in thousandths of an inch

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, PrintRect returns null.

Usage
The PrintOval, PrintRect, and PrintRoundRect functions draw filled shapes. To print other shapes or text inside the shapes, draw the filled shape first and then add text and other shapes or lines inside it. If you draw the filled shape after other printing functions, it will cover anything inside it. For example, to draw a border around text and lines, draw the oval or rectangular border first and then use PrintLine and PrintText to position the lines and text inside.

PrintRect does not change the position of the print cursor, which remains where it was before the function was called. In general, print functions in which you specify coordinates do not affect the print cursor.

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:

```power
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

See also
PrintBitmap
PrintClose
PrintLine
PrintOpen
PrintOval
PrintRoundRect

PrintRoundRect

Description
Draws a white rectangle with rounded corners and a border of the specified thickness on the print page.

Syntax
PrintRoundRect ( printjobnumber, x, y, width, height, xradius, yradius, thickness )

Argument | Description
--- | ---
printjobnumber | The number the PrintOpen function assigned to the print job
x | An integer specifying the x coordinate in thousandths of an inch of the upper-left corner of the rectangle
y | An integer specifying the y coordinate in thousandths of an inch of the upper-left corner of the rectangle
width | An integer specifying the rectangle’s width in thousandths of an inch
height | An integer specifying the rectangle’s height in thousandths of an inch
xradius | An integer specifying the x radius of the corner rounding
yradius | An integer specifying the y radius of the corner rounding
thickness | An integer specifying the thickness of the rectangle’s border line in thousandths of an inch

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, PrintRoundRect returns null.

Usage
The PrintOval, PrintRect, and PrintRoundRect functions draw filled shapes. To print other shapes or text inside the shapes, draw the filled shape first and then add text and other shapes or lines inside it. If you draw the filled shape after other printing functions, it will cover anything inside it. For example, to draw a border around text and lines, draw the oval or rectangular border first and then use PrintLine and PrintText to position the lines and text inside.
PrintRoundRect does not change the position of the print cursor, which remains where it was before the function was called. In general, print functions in which you specify coordinates do not affect the print cursor.

Examples

This example starts a new print job, which begins a new page, and prints a rectangle with rounded corners as a page border. Then it closes the print job, which sends the page to the printer.

The rectangle is 6 1/4 inches wide by 9 inches high and its upper corner is one inch from the top and one inch from the left edge of the print area. The border has a line thickness of 1/8 of an inch and the corner radius is 300:

```
long Job

    // Define a new blank page.
    Job = PrintOpen()

    // Print a RoundRectangle on the page.
    PrintRoundRect (Job, 1000,1000, 6250,9000, & 300,300, 125)

    // Send the page to the printer.
    PrintClose(Job)
```

See also
PrintBitmap
PrintClose
PrintLine
PrintOpen
PrintOval
PrintRect

---

**PrintScreen**

Description
Prints the screen image as part of a print job.

Syntax

```
PrintScreen ( printjobnumber, x, y {, width, height} )
```

<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
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:

```powerscript
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.
*/
```

<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>string</code></td>
<td>A string you want to send to the printer. In the string, use ASCII values for nonprinting characters.</td>
</tr>
<tr>
<td><code>zerochar</code> (optional)</td>
<td>An ASCII value (1 to 255) that you want to use to represent the number zero in string.</td>
</tr>
</tbody>
</table>
PrintSetFont

**PrintSend(Job," ~ h1BX ~ 030 ~ 040")**
... // Print text or DataWindow

// Send the job to the printer or spooler.
PrintClose(Job)

This example opens a print job and sends an escape sequence to a printer in IBM Proprinter mode to change the margins. The decimal ASCII code 255 represents zero:

```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

---

**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**

```powerbuilder
PrintSetFont ( printjobnumber, fontnumber )
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>printjobnumber</em></td>
<td>The number the <em>PrintOpen</em> function assigned to the print job</td>
</tr>
<tr>
<td><em>fontnumber</em></td>
<td>The number (1 to 8) of a font defined for the job in <em>PrintDefineFont</em> or 0 (the default font for the printer)</td>
</tr>
</tbody>
</table>
Return value

Integer. Returns the character height of the current font if it succeeds and -1 if an error occurs. If any argument’s value is null, PrintSetFont returns null.

Examples

This example starts a new print job and specifies that font number 2 is Courier, 18 point, bold, default pitch, in modern font, with no italic or underline. The PrintSetFont statement sets the current font to font 2. Then the Print statement prints the company name:

```
long Job

// Start a new print job and a new page.
Job = PrintOpen()

// Define the font for Job.
PrintDefineFont(Job, 2, "Courier 10Cps", & 250, 700, Default!, Modern!, FALSE, FALSE)

// Set the font for Job.
PrintSetFont(Job, 2)

// Print the company name in the specified font.
Print(Job,"Sybase Corporation")
```

See also

PrintDefineFont
PrintOpen

---

**PrintSetPrinter**

**Description**

Sets the printer to use for the next print function call. This function does not affect open jobs.

**Syntax**

```
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:

```
long ll_place
```
**PrintSetSpacing**

Sets the factor that PowerBuilder uses to calculate line spacing.

**Syntax**

```powerbuilder
PrintSetSpacing(printjobnumber, spacingfactor)
```

<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>spacingfactor</td>
<td>The number by which you want to multiply the character height to determine the vertical line-to-line spacing. The default is 1.2.</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, 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):

```powerbuilder
long Job

// Define a new blank page.
Job = PrintOpen()

// Set the spacing factor.
PrintSetSpacing(Job, 1.5)
```

See also

PrintOpen
PrintSetup

Description
Calls the Printer Setup dialog box provided by the system printer driver and lets the user specify settings for the printer.

Syntax
PrintSetup()

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

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:

```power
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.

```power
long ll_place
string ls_setptr
string ls_prntrs = PrintGetPrinters()

ll_place=lastpos(ls_prntrs, "~n")
```
PrintText

Description
Prints a single line of text starting at the specified coordinates.

Syntax
PrintText ( printjobnumber, string, x, y [{, fontnumber} ] )

Argument | Description
--- | ---
printjobnumber | The number the PrintOpen function assigned to the print job.
string | A string whose value is the text you want to print.
x | An integer specifying the x coordinate in thousandths of an inch of the beginning of the text.
y | An integer specifying the y coordinate in thousandths of an inch of the beginning of the text.
fontnumber | (optional) The number (1 to 8) of a font defined for the job by using the PrintDefineFont function or 0 (the default font for the printer). If you omit fontnumber, the text prints in the current font for the print job.

Return value
Integer. Returns the x coordinate of the new cursor location (that is, the value of the parameter x plus the width of the text) if it succeeds. PrintText returns -1 if an error occurs. If any argument’s value is null, PrintText returns null.

Usage
PrintText does change the position of the print cursor, unlike the other print functions for which you specify coordinates. The print cursor moves to the end of the printed text. PrintText also returns the x coordinate of the print cursor. You can use the return value to determine where to begin printing additional text.

PrintText does not change the print cursor’s y coordinate, which is its vertical position on the page.

Examples
These statements start a new print job and then print PowerBuilder in the current font 3.7 inches from the left edge at the top of the page (location 3700,10):

```plaintext
long Job
```

See also
PrintGetPrinter
PrintSetPrinter
PrintSetup

```plaintext
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 )

Argument     | Description
--------------|-------------
printjobnumber | The number the PrintOpen function assigned to the print job
string        | A string whose value is the text for which you want to determine the width

Return value
Integer. Returns the width of string in thousandths of an inch using the current font of 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:

```plaintext
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")
```
PrintX

Description
Reports the x coordinate of the print cursor.

Syntax
PrintX( printjobnumber )

Argument Description
printjobnumber The number the PrintOpen function assigned to the print job

Return value
Integer. Returns the x coordinate of the print cursor if it succeeds and -1 if an error occurs. If any argument’s value is null, PrintX returns null.

Examples
These statements set LocX to the x coordinate of the cursor and print "End of Report" an inch beyond that location:

```power
integer LocX
long Job

Job = PrintOpen()
... //Print statements
LocX = PrintX(Job)
Print(LocX+1000, "End of Report")
```

See also
PrintY

PrintY

Description
Reports the y coordinate of the print cursor.

Syntax
PrintY( printjobnumber )

Argument Description
printjobnumber The number the PrintOpen function assigned to the print job

Return value
Integer. Returns the y coordinate of the print 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:

```powershell
integer LocX, LocY
long Job
Job = PrintOpen()
... //Print statements
LocX = PrintX(Job)
LocY = PrintY(Job) + 1000
PrintBitmap(Job, "CORP.BMP", LocX, LocY, 1000,1000)
```

See also

PrintX

### ProfileInt

**Description**

Obtains the integer value of a setting in the profile file for your application.

**Syntax**

```powershell
ProfileInt ( filename, section, key, default )
```

**Argument**

<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 specify a full path, ProfileInt uses the operating system's standard file search order to find the file.</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. In the file, section names are in square brackets. Do not include the brackets in <code>section</code>. <code>Section</code> is not case sensitive.</td>
</tr>
<tr>
<td><code>key</code></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>. <code>Key</code> is not case sensitive.</td>
</tr>
<tr>
<td><code>default</code></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:

```
[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:

```powerShell
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:

```powerShell
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
Rand ( n )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</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.

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:

```power_script
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:

```power_script
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

Description

Reads data from an OLE stream object into a string.

Applies to

OLEStream objects

Syntax

```power_script
olestream.Read { variable {, stopforline } }
```
Read

### Argument | Description
--- | ---
olestream | The name of an OLE stream variable that has been opened.
variable | The name of a string variable into which want to read data from olestream.
stopforline (optional) | A boolean value that specifies whether to read a line at a time. In other words, Read will stop reading at the next carriage return/linefeed. Values are:
- TRUE – (Default) Stop at the end of a line and leave the read pointer positioned after the carriage return/linefeed so the next read will read the next line
- FALSE – Read the whole stream or a maximum of 32,765 bytes

### 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
CHAPTER 10  PowerScript Functions

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

Syntax

\texttt{olestream.Read( variable\{, maximumread\})}

<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{variable}</td>
<td>The name of a blob variable or character array into which want to read data from \texttt{olestream}.</td>
</tr>
<tr>
<td>\texttt{maximumread}</td>
<td>A long value specifying the maximum number of bytes to be read. The default is 32,765 or the length 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
-2 Read error
-9 Other error

If any argument’s value is \texttt{null}, \texttt{Read} returns \texttt{null}.

Examples
This example opens an OLE object in the file \textit{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 reads the contents of \texttt{olestr\_info} into the blob \texttt{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:

\begin{verbatim}
boolean lb_memexists
OLEStorage stg_stuff
OLEStream olestr_info
blob lb_info

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

olestr_info.Open(stg_stuff, "info", &
    stgRead!, stgExclusive!)
olestr_info.Read(lb_info)
\end{verbatim}

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

\[
\text{Real}(\text{stringorblob})
\]

<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 \text{stringorblob} as a real. If \text{stringorblob} is not a valid PowerScript number or is an incompatible datatype, \text{Real} returns 0. If \text{stringorblob} is null, \text{Real} returns null.

Examples

This statement returns 24 as a real:

\[
\text{Real}("24")
\]

This statement returns the contents of the SingleLineEdit sle_Temp as a real:

\[
\text{Real}(\text{sle_Temp.Text})
\]

The following example, although of no practical value, illustrates how to assign real values to a blob and how to use \text{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 \text{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 \text{Real} function knows how many bytes to evaluate:

\[
\begin{align*}
\text{blob}\{20\} & \text{ lb_blob} \cr
\text{real} & \text{ r1, r2} \cr
\text{integer} & \text{ len1, len2} \cr
\text{len1} & = \text{BlobEdit(\text{lb_blob}, 1, 32750E0)} \cr
\text{len2} & = \text{BlobEdit(\text{lb_blob}, \text{len1, 43750E0})} \cr
\text{// Extract the real value at the beginning and} \cr
\text{// ignore the rest of the blob} \cr
\text{r1} & = \text{Real(\text{lb_blob})} \cr
\text{// Extract the second real value stored in the blob} \cr
\text{r2} & = \text{Real(\text{BlobMid(\text{lb_blob, len1, len2 - len1})})}
\end{align*}
\]

See also

Double
Integer
Long
Real method for DataWindows in the DataWindow Reference or online Help
**RecognizeText**

**Description**
Specifies that text in an InkEdit control should be recognized.

**Applies to**
InkEdit controls

**Syntax**

```power_script
inkeditname.RecognizeText()
```

**Argument** | **Description**
---|---
`inkeditname` | The name of the InkEdit control in which you want to recognize text.

**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:

```power_script
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**

```power_script
RegistryDelete( key, valuenname )
```

**Argument** | **Description**
---|---
`key` | A string whose value is the key in the system registry you want to delete or whose value you want to delete. To uniquely identify a key, specify the list of parent keys above it in the hierarchy, starting with the root key. The keys in the list are separated by backslashes.

`valuenname` | 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.

**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:

```powershell
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

```powershell
RegistryGet ( key, valuename, { valuetype }, valuevariable )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>key</code></td>
<td>A string whose value names the key in the system registry whose value you want. To uniquely identify a key, specify the list of parent keys above it in the hierarchy, starting with the root key. The keys in the list are separated by backslashes.</td>
</tr>
<tr>
<td><code>valuename</code></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>
</tbody>
</table>
| `valuetype`   | A value of the RegistryValue enum identifying the datatype of a value in the registry. Values are:
  - RegString! – A null-terminated string
  - RegExpandString! – A null-terminated string that contains unexpanded references to environment variables
  - RegBinary! – Binary data
  - ReguLong! – A 32-bit number
  - ReguLongBigEndian! – A 32-bit number
  - RegLink! – A Unicode symbolic link
  - RegMultiString! – An unbounded array of strings |
| `valuevariable` | A variable corresponding to the datatype of `valuetype` in which you want to store the value obtained from the system registry for the specified key and value name. |
Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. An error is returned if the 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 filename 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:

```power
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:

```power
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.

Syntax

```power
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>
</tbody>
</table>
**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**

```powershell
RegistrySet (key, valuename, valuetype, value)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>subkeys</code></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`:

```powershell
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
### 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.

<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. To uniquely identify a key, specify the list of parent keys above it in the hierarchy, starting with the root key. The keys in the list are separated by backslashes. If <code>key</code> does not exist in the registry, <code>RegistrySet</code> creates a new key. To create a key without a named value, specify an empty string for <code>valuenname</code>.</td>
</tr>
<tr>
<td>valuenname</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. If <code>valuenname</code> does not exist in the registry, <code>RegistrySet</code> causes a new name to be created for <code>key</code>.</td>
</tr>
</tbody>
</table>
| valuetype | A value of the RegistryValue enumerated datatype identifying the datatype of a value in the registry. Values are:  
  - RegString! – A null-terminated string  
  - RegExpandString! – A null-terminated string that contains unexpanded references to environment variables  
  - RegBinary! – Binary data  
  - ReguLong! – A 32-bit number  
  - ReguLongBigEndian! – A 32-bit number  
  - RegLink! – A Unicode symbolic link  
  - RegMultiString! – An unbounded array of strings |
| value    | A variable corresponding to the datatype of `valuetype` containing a value to be set in the registry. |
RegistrySet

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
</table>
| Key        | An element in the registry. A key is part of a tree of keys, descending from one of the predefined root keys. Each key is a subkey or child of the parent key above it in the hierarchy. There are four root strings:  
  • HKEY_CLASSES_ROOT  
  • HKEY_LOCAL_MACHINE  
  • HKEY_USERS  
  • HKEY_CURRENT_USER  
  A key is uniquely identified by the list of parent keys above it. The keys in the list are separated by slashes, as shown in these examples:  
    HKEY_CLASSES_ROOT\Sybase.Application  
    HKEY_USERS\MyApp\Display\Fonts |
| Value name | The name of a value belonging to the key. A key can have one unnamed value and one or more named values. |
| Value type | A value identifying the datatype of a value in the registry.                   |
| Value      | A value associated with a value name or an unnamed value. Several string, numeric, and binary datatypes are supported by 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!, sie_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**

Obtains the list of named values associated with a key.

**Syntax**

\[ \text{RegistryValues} \left( \text{key}, \text{valuename} \right) \]

**Argument** | **Description**  
---|---  
key | A string whose value is the key in the system registry for which you want the values of its subkeys.  
To uniquely identify a key, specify the list of parent keys above it in the hierarchy, starting with the root key. The keys in the list are separated by backslashes.  
valuename | An array variable of strings in which you want to store the names.  
If the array is variable size, its upper bound will reflect the number of named values found.  
If the array is fixed size, it must be large enough to hold all the names. However, there will be no way to know how many names were actually found.

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

**Usage**  
For more information about entries in the system registry, see `RegistrySet`.

**Examples**  
This example gets the value names associated with the key `Fonts` and stores them in the array `ls_valuearray`:

```powerShell
string ls_valuearray[]
RegistryValues( &
"HKEY_LOCAL_MACHINE\Software\MyApp.Settings\Fonts", &ls_valuearray)
```

**See also**  
RegistryDelete  
RegistryGet  
RegistryKeys  
RegistrySet
RelativeDate

Description
Obtains the date that occurs a specified number of days after or before another date.

Syntax
RelativeDate (date, n)

Argument | Description
--- | ---
date | A value of type date
n | An integer indicating a number of days

Return value
Date. Returns the date that occurs n days after date if n is greater than 0. Returns the date that occurs n days before date if n is less than 0. If any argument’s value is null, RelativeDate returns null.

Examples
This statement returns 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.

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
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 )`

Arguments and Description:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>olename</code></td>
<td>The name of the OLE control containing the object for which you want the native pointer.</td>
</tr>
<tr>
<td><code>pointer</code></td>
<td>A UnsignedLong variable that holds the pointer you want to release. ReleaseNativePointer sets <code>pointer</code> 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 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

Description
Removes a directory.

Syntax
`RemoveDirectory ( directoryname )`

Arguments and Description:
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>directoryname</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```plaintext
string ls_path="my targets"
integer li_filenum

li_filenum = 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 )`

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.
**Replace**

**Usage**

When errors have occurred during a pipeline data transfer, Start populates its pipeline-error DataWindow control with the rows that caused the errors. The user or a script can then make corrections to the data. The Repair function is usually associated with a CommandButton that the user can click after correcting data in the pipeline-error DataWindow.

If errors occur again, the rows that are in error remain in the pipeline-error DataWindow. The user can correct the data again and click the button that calls Repair.

**Examples**

This statement connects to the destination database using the transaction instance variable `i_dst`. It then updates the database with the corrections made in the Error DataWindow for pipeline `i_pipe`:

```
i_pipe.Repair(i_dst)
```

**See also**

Cancel
Repair
Start

---

**Replace**

**Description**

Replaces a portion of one string with another.

**Syntax**

```
Replace ( string1, start, n, string2 )
```

**Argument** | **Description**
---|---
`string1` | The string in which you want to replace characters with `string2`.
`start` | A long whose value is the number of the first character you want replaced. (The first character in the string is number 1.)
`n` | A long whose value is the number of characters you want to replace.
`string2` | The string that will replace characters in `string1`. The number of characters in `string2` can be greater than, equal to, or less than the number of characters you are replacing.

**Return value**

String. Returns the string with the characters replaced if it succeeds and the empty string if it fails. If any argument’s value is null, Replace returns null.

**Usage**

If the start position is beyond the end of the string, Replace appends `string2` to `string1`. If there are fewer characters after the start position than specified in `n`, Replace replaces all the characters to the right of character `start`.

If `n` is zero, then, in effect, Replace inserts `string2` into `string1`.
Examples

These statements change the value of `Name` from Davis to Dave:

```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))
```
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.

See also
Replace method for DataWindows in the DataWindow Reference or the online Help.

LOOP
    sle_1.Text = mystring

872 PowerBuilder
ReplaceText

Description
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)

Arguments
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editname</td>
<td>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.</td>
</tr>
<tr>
<td>string</td>
<td>The string that replaces the selected text.</td>
</tr>
</tbody>
</table>

Return value
Integer for DataWindow, InkEdit, and list boxes, Long 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 `ddlb_Actions`:

```
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

```
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`:

```
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 ( )

<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
ErrorReturn. Returns one of the following values:

- Success! – The function succeeded
- FileNotOpenError! – The specified trace file has not been opened

Usage
Use this function to return to the start of the open trace file and begin rereading the contents of the file. To use the Reset function, you must have previously opened the trace file with the Open function. You use the Reset and Open functions as well as the other properties and functions provided by the TraceFile object to access the contents of a trace file directly. You use these functions if you want to perform your own analysis of the tracing data instead of using the available modeling objects.

Examples
This example returns execution to the start of the open trace file ltf_file so that the file’s contents can be reread:

```
TraceFile ltf_file
string ls_filename

ltf_file = CREATE TraceFile
ltf_file.Open(ls_filename)
...
ltf_file.Reset(ls_filename)
...
```

See also
Open
NextActivity
Close
CHAPTER 10  PowerScript Functions

ResetArgElements

Description  Clears the argument list.

Applies to  Window ActiveX controls

Syntax  

**activexcontrol**.**ResetArgElements**()  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>activexcontrol</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```vbscript  
...  
retcd = PBRX1.TriggerPBEvent(theEvent, numargs)  
rc = PBRX1.GetLastReturn()  
IF rc <> 1 THEN  
msgbox "Error. Empty string."  
END IF  
PBRX1.**ResetArgElements**();  
...  
```

See also  

**GetLastReturn**  
**InvokePBFunction**  
**SetArgElement**  
**TriggerPBEvent**
**ResetDataColors**

**Description**
Restores the color of a data point to the default color for its series.

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

**Syntax**
```
controlname.ResetDataColors( { graphcontrol, } 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 reset the color of a data point, or the name of the DataWindow 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 reset the color</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number of the series in which you want to reset the color of a data point</td>
</tr>
<tr>
<td>datapointnumber</td>
<td>The number of the data point for which you want to reset the color</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, ResetDataColors returns null.

**Default color for data points**
To set the color for a series, use SetSeriesStyle. The color you set for the series is the default color for all data points in the series.

**Examples**
These statements change the color of data point 10 in the series named Costs in the graph `gr_product_data` to the color for the series:
```
SeriesNbr = gr_product_data.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:
```
SeriesNbr = dw_equipment.FindSeries("Costs")
dw_equip.ResetDataColors("gr_comps", SeriesNbr, 10)
```

**See also**
GetDataStyle
SeriesName
GetSeriesStyle
SetDataStyle
SetSeriesStyle
ResetInk
Description
Clears ink from an InkPicture control.
Applies to
InkPicture controls
Syntax

\[
\text{inkpicname}.\text{ResetInk}()
\]

Return value
Integer. Returns 1 for success and -1 for failure.
Usage
Use the \text{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

\[
\text{inkpicname}.\text{ResetPicture}()
\]

Return value
Integer. Returns 1 for success and -1 for failure.
Usage
Use the \text{ResetPicture} function to clear the image from an InkPicture control.
Examples
The following example clears the image from an InkPicture control:

\[
\text{ip}_1.\text{ResetPicture}()
\]
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 )
```

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:

\[ \text{gb\_box1.\textit{Resize}(100, 150)} \]

This statement doubles the width and height of the picture control p_1:

\[ \text{p\_1.\textit{Resize}(p\_1.Width*2, p\_1.Height*2)} \]

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
\[ \text{jaguarorb.\textit{Resolve\_Initial\_References}(\text{objstring}, \text{object})} \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jaguarorb</td>
<td>An instance of JaguarORB</td>
</tr>
<tr>
<td>objstring</td>
<td>A string that has the value &quot;NameService&quot;</td>
</tr>
<tr>
<td>object</td>
<td>A reference variable of type CORBAObject that will contain a reference to the COS naming service</td>
</tr>
</tbody>
</table>

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 \textit{Resolve\_Initial\_References} function to obtain the initial naming context. However, this technique is not recommended because it requires use of a deprecated SessionManager::Factory 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 \textit{CreateInstance} and \textit{Lookup} functions of the Connection object.

You can also use the JaguarORB object’s \textit{String\_To\_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.
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:

```plaintext
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://server1:9000'")

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

\[
\text{RespondRemote ( boolean )}
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>A boolean expression. \text{true} indicates that the previously received command or data was acceptable. \text{false} indicates that it was not.</td>
</tr>
</tbody>
</table>

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, \text{RespondRemote} returns null.

Usage You can use \text{RespondRemote} when the PowerBuilder application is the DDE server or DDE client application.

You usually call \text{RespondRemote} after these functions:

\begin{itemize}
  \item GetCommandDDE
  \item GetCommandDDEOrigin
  \item GetDataDDE
  \item GetDataDDEOrigin
\end{itemize}

For more information about PowerBuilder as a client, see \text{OpenChannel} and \text{ExecRemote}. For more information about PowerBuilder as a server, see \text{StartServerDDE}.

Examples In a script for the HotLinkAlarm event, these statements tell a remote application named Gateway that its data was successfully received:

\[
\begin{align*}
\text{String Applname, Topic, Item, Value} \\
\text{GetDataDDEOrigin(Applname, Topic, Item)} \\
\text{IF Applname = "Gateway" THEN}
\end{align*}
\]
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 135

ResumeTransaction

**Description**
Associates the EAServer transaction passed as an argument with the calling thread.

**Applies to**
CORBACurrent objects

**Syntax**
```
CORBACurrent.ResumeTransaction (handletrans)
```
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 the `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 `ll_handle` argument returned by `SuspendTransaction`:

```power
// Instance variable:
// CORBACurrent corbcurr
integer li_rc
unsignedlong ll_handle

li_rc = this.GetContextService("CORBACurrent", & corbcurr)
li_rc = corbcurr.Init()
li_rc = corbcurr.BeginTransaction()
// do some transactional work
ll_handle = corbcurr.SuspendTransaction()
//do some non-transactional work
li_rc = corbcurr.ResumeTransaction(ll_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 )

Argument | Description
--- | ---
string | A string whose characters you want to reorder so that the last character is first and the first character is last

Return value
String. Returns a string with the characters of string in reversed order. Returns the empty string if it fails.

Usage
Reverse is useful with the IsArabic and IsHebrew functions, which help you implement 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:

```plaintext
string ls_name
ls_name = Reverse(sle_name.Text)
```

See also
IsArabic
IsHebrew
CHAPTER 10  PowerScript Functions

RevertToSelf

Description  Restores the security attributes for a COM object that is running on COM+ and impersonating the client.

Applies to  TransactionServer objects

Syntax  

\[ \text{transactionserver}.\text{RevertToSelf}() \]

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:

```
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  

\[ \text{RGB}( \text{red}, \text{green}, \text{blue} ) \]
RGB

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:

\[ 65536 \times \text{Blue} + 256 \times \text{Green} + \text{Red} \]

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>
<tr>
<td>Green</td>
<td>0</td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td>Dark Green</td>
<td>0</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>Blue</td>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>Dark Blue</td>
<td>0</td>
<td>0</td>
<td>128</td>
</tr>
</tbody>
</table>
CHAPTER 10  PowerScript Functions

Examples

This statement returns a long that represents black:

\texttt{RGB(0, 0, 0)}

This statement returns a long that represents white:

\texttt{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{verbatim}
  st_title.TextColor = RGB(0, 255, 0)
  st_title.BackColor = RGB(128, 0, 128)
\end{verbatim}

See also\texttt{RGB} method for DataWindows in the DataWindow Reference or the online Help

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Color & Red value & Green value & Blue value \\
\hline
Magenta & 255 & 0 & 255 \\
Dark Magenta & 128 & 0 & 128 \\
Cyan & 0 & 255 & 255 \\
Dark Cyan & 0 & 128 & 128 \\
Yellow & 255 & 255 & 0 \\
Brown & 128 & 128 & 0 \\
\hline
\end{tabular}
\end{table}

\section*{Right}

\textbf{Description}

Obtains a specified number of characters from the end of a string.

\textbf{Syntax}

\texttt{Right ( string, n )}

\begin{center}
\begin{tabular}{|c|c|}
\hline
Argument & Description \\
\hline
\textit{string} & The string from which you want characters returned \\
\textit{n} & A long whose value is the number of characters you want returned from the right end of \textit{string} \\
\hline
\end{tabular}
\end{center}

\textbf{Return value}

String. Returns the rightmost \textit{n} characters in \textit{string} if it succeeds and the empty string ("") if an error occurs. If any argument’s value is null, \texttt{Right} returns null. If \textit{n} is greater than or equal to the length of the string, \texttt{Right} returns the entire string. It does not add spaces to make the return value’s length equal to \textit{n}.

\textbf{Examples}

This statement returns RUTH:

\begin{verbatim}
  Right("BABE RUTH", 4)
\end{verbatim}
This statement returns BABE RUTH:

RightA("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.

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 )
CHAPTER 10  PowerScript Functions

RightTrim
Description  Removes spaces from the end of a string.
Syntax  
RightTrim (string)
RightTrimW (string)

<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>
</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.

Examples  This statement returns RUTH:

```
RightTrim("RUTH ")
```

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  
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  
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>
**RollbackOnly**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Integer. Returns 0 if it succeeds and one of the following negative values if an error occurs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Failed for unknown reason</td>
</tr>
<tr>
<td>-2</td>
<td>No transaction is associated with the calling thread</td>
</tr>
</tbody>
</table>

**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:

```c++
// 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()

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

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:

```powerscript
// Instance variables:
// CORBACurrent corbcurr
int li_rc1, li_rc2
long ll_rc

this.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
```
### Round

**Description**
Rounds a number to the specified number of decimal places.

**Syntax**

```plaintext
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:

```plaintext
Round(9.624, 2)
```

This statement returns 9.63:

```plaintext
Round(9.625, 2)
```
This statement returns 9.600:
\[
\text{Round}(9.6, 3)
\]
This statement returns –9.63:
\[
\text{Round}(-9.625, 2)
\]
This statement returns null:
\[
\text{Round}(-9.625, -1)
\]

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**
\[
\text{instancename}.\text{RoutineList}(\text{list})
\]

<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 <code>RoutineList</code> 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:

```powershell
Long ll_cnt
ProfileCall lproc_call[]

lpro_model.BuildModel()
lpro_model.RoutineList(iprotn_list)
...
```

See also

ClassList

---

**Run**

**Description**

Runs the specified application program.

**Syntax**

```powershell
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! – Maximised; enlarge the program window to its maximum size when it starts
- Minimized! – Minimised; 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.
CHAPTER 10  PowerScript Functions

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 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>Syntax 1</th>
<th>Syntax 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>InkPicture controls</td>
<td>OLE controls</td>
</tr>
</tbody>
</table>

**Syntax 1**

**Description**

Saves a picture and optionally overlay ink to a file or blob from an InkPicture control.

**Applies to**

InkPicture controls

**Syntax**

\[ \text{inkpicname}.\text{Save}( t | b , \text{format} \{ , \text{WithInk} \} ) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inkpicname</td>
<td>The name of the InkPicture control from which you want to save a picture.</td>
</tr>
<tr>
<td>t</td>
<td>A string containing the name and location of the file into which the picture will be saved.</td>
</tr>
<tr>
<td>b</td>
<td>The name of a blob passed by reference that will hold the picture in the control.</td>
</tr>
<tr>
<td>format</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>WithInk (optional)</td>
<td>A boolean specifying whether overlay ink should be saved with the picture. Values are: True – overlay ink is saved with the picture (default) False – 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.
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:

```powerscript
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:

```powerscript
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
Save

Syntax 2

Description

Saves an OLE object in an OLE control or an OLE storage object.

Syntax

\texttt{oleobject.Save()}  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{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 \texttt{oleobject} is null, \texttt{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 \texttt{Open} function. When you call \texttt{SaveAs}, the old connection is ended and a new connection is established with another storage or file.

When you call \texttt{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 \texttt{Save} function in PowerBuilder returns an error.

When you call \texttt{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 \texttt{Open} or \texttt{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 \texttt{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 \texttt{ole_1} back to the storage from which it was loaded, either a storage object variable or a file on disk:

\begin{verbatim}
integer result
result = ole_1.Save()
\end{verbatim}
This example saves a storage object to its file. `Olestor_1` is an instance variable of type `olestorage`:

```powerShell
integer result
result = olestor_1.Save()
```

In a window’s Open script, this code creates a storage variable `ole_stor`, which is declared as an instance variable, and associates it with a storage file that contains several Visio drawings. The script then opens one of the drawings into the control `ole_draw`. After the user activates and edits the object, the script for a Save button saves the object to the storage and then to the storage’s file.

The script for the window’s Open event includes:

```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

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.

**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**

`controlname.SaveAs ( { filename, } { graphcontrol, saveastype, colheading { , encoding } } )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>controlname</code></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><code>filename</code></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 <code>filename</code> or specify an empty string (&quot;&quot;), PowerBuilder prompts the user for a file name.</td>
</tr>
<tr>
<td><code>graphcontrol</code></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>
</tbody>
</table>
### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| `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!.  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>colheading</code> (optional)</td>
<td>A boolean value indicating whether you want column headings with the saved data. The default value is <code>true</code>. <code>Colheading</code> is ignored for dBASE files; column headings are always saved.</td>
</tr>
</tbody>
</table>
SaveAs

**Argument**  
*encoding* (optional)  
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:
- EncodingANSI! (default)
- EncodingUTF8!
- EncodingUTF16LE!
- EncodingUTF16BE!

**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:

``` Powershell
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:

``` Powershell
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:

``` Powershell
dw_equipmt.SaveAs("gr_computers", &
"G:\INVENTORY\SALES.XLS", Excel!, TRUE)
```

**See also**  
Print
### Syntax 2

**For saving an OLE control to a file**

Saves the object in an OLE control in a storage file.

**Applies to** OLE controls

**Syntax**

```
olecontrol.SaveAs (OLEtargetfile)
```

<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>OLEtargetfile</td>
<td>A string specifying the name of an OLE storage file. The file can already exist. OLEtargetfile can include a path, as well as information about where to store the object in the file’s internal structure.</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`:

```
integer result  
result = ole_1.SaveAs("c:\ole\expense.ole")
```

**See also**

Open  
Save
**SaveAs**

**Syntax 3**

**For saving an OLE control to an OLE storage**

Saves the object in an OLE control to an OLE storage object in memory.

**Applies to**

OLE controls

**Syntax**

`olecontrol.SaveAs (targetstorage, substoragename)`

<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 containing the object you want to save.</td>
</tr>
<tr>
<td><code>targetstorage</code></td>
<td>The name of an object variable of OLEStorage in which to store the object in <code>olecontrol</code>.</td>
</tr>
<tr>
<td><code>substoragename</code></td>
<td>A string whose value is the name of a substorage within <code>targetstorage</code>. If <code>substorage</code> does not exist, <code>SaveAs</code> 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`:

```powerbuilder
integer result
result = ole_1.SaveAs(stg_stuff)
```

**See also**

Open  
Save
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)`

<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 OLE object you want to save.</td>
</tr>
<tr>
<td><code>OLEtargetfile</code></td>
<td>A string specifying the name of a new OLE storage file. <code>OLEtargetfile</code> 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:

```plaintext
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><code>olestorage</code></td>
<td>The name of an object variable of type OLEStorage containing the OLE object you want to save.</td>
</tr>
<tr>
<td><code>substoragename</code></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><code>targetstorage</code></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>
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:

```
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**

```
rtename.SaveDocument ( filename {, filetype {, encoding }} )
```

**Argument** | **Description**
--- | ---
rtename | The name of the RichTextEdit control whose contents you want to save.
SaveDocument

<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 to be saved. If the file already exists, the FileExists event is triggered.</td>
</tr>
</tbody>
</table>
| filetype (optional) | A value of the FileType enumerated datatype specifying the format of the saved file. Values are:  
  - FileTypeRichText! – Save the file in rich text format  
  - FileTypeText! – Save the file as text  
  - FileTypeDoc! – Save the file in Microsoft Word format  
  - FileTypeHTML! – Save the file in HTML format  
  - FileTypePDF! – Save the file in PDF format |
| encoding (optional) | Character encoding of the file to which the data is saved. This parameter applies only to text files. If you do not specify an encoding parameter, the file is saved in ANSI format.  
The filetype argument must be set to FileTypeText! If the filetype argument is set to any other file type, this argument is ignored.  
Values are:  
  - EncodingANSI! (default)  
  - EncodingUTF8!  
  - EncodingUTF16LE!  
  - EncodingUTF16BE! |

Return value

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

Usage

SaveDocument triggers a FileExists event when the file you name already exists. If you do not specify a filetype, 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 encoding argument is valid only if you specified FileTypeText! for the filetype argument. SaveDocument saves text in ANSI format only for all other file types.

Examples

This code for a CommandButton saves the document in the RichTextEdit rte_1:

```powershell
integer li_rtn
li_rtn = rte_1.SaveDocument("c:\test.rtf", &FileTypeRichText!)
```

If the file TEST.RTF already exists, PowerBuilder triggers the FileExists event with the following script. OpenWithParm displays a response window that asks the user if it is OK to overwrite the file. The return value from FileExists determines whether the file is saved:
OpenWithParm( w_question, &
   "The specified file already exists. " + &
   "Do you want to overwrite it?" )
IF Message.StringParm = "Yes" THEN
   RETURN 0 // File is saved
ELSE
   RETURN -1 // Saving is canceled
END IF

This code for a CommandButton saves the document in the RichTextEdit rte_1
in a text file with UTF-16LE encoding:

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**

```
inkpicname.SaveInk( t | b {, format {, mode } } )
```

**Argument** | **Description**
--- | ---
inkpicname | The name of the InkPicture control from which you want to save ink.
t | A string containing the name and location of a file that will hold the ink you want to save from the control.
b | The name of a blob passed by reference that will hold the ink you want to save from the control.
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)
**SaveInk**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mode</code> (optional)</td>
<td>A value of the InkCompressionMode enumerated variable that specifies the compression mode in which you want to save the ink. Values are:&lt;br&gt;• DefaultCompression! (default)&lt;br&gt;• MaximumCompression!&lt;br&gt;• NoCompression!</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:

```powershell
int li_return
string ls.pathname, ls.filename

GetFileSaveName("Save As", ls.pathname, ls.filename, "ISF")
li_return = ip_1.SaveInk(ls.pathname)
```

The following example saves the ink in an InkPicture control into a GIF file with maximum compression:

```powershell
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)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>editname</code></td>
<td>The name of the DataWindow, RichTextEdit, or MultiLineEdit in which you want to scroll up or down. If <code>editname</code> is a DataWindow, then Scroll affects its edit control.</td>
</tr>
<tr>
<td><code>number</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

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

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 $rtename$.ScrollNextRow ( )

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
rename.ScrollPriorPage ()

Argument | Description
---|---
rename | The name of the RichTextEdit or DataWindow control in which you want to scroll to the prior page.

The DataWindow object in the DataWindow control must be a RichTextEdit DataWindow.

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

Usage
When the RichTextEdit shares data with a DataWindow, the RichTextEdit contains multiple instances of the document, one instance for each row.

When the first page of the document for one row is visible, calling ScrollPriorPage goes to the last page for the prior row.

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:

rte_1.ScrollPriorPage ()

See also
Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorRow
ScrollPriorRow

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

```plaintext
rtename.ScrollPriorRow()
```

<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 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.

```plaintext
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><code>rtename</code></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><code>row</code></td>
<td>A long identifying the row to which you want to scroll. If <code>row</code> is 0, <code>ScrollToRow</code> scrolls to the first row. If <code>row</code> 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.

**Usage**

When the RichTextEdit shares data with a DataWindow, the RichTextEdit contains multiple instances of the document, one instance for each row. `ScrollToRow` goes to the instance associated with the specified row.

**Examples**

In this example, `dw_1` has retrieved at least 25 rows of data. After calling `DataSource`, the RichTextEdit control contains at least 25 instances of its document. `ScrollToRow` scrolls to the 25th instance:

```
rte_1.DataSource(dw_1)
rte_1.ScrollToRow(25)
```

**See also**

Scroll

ScrollNextPage

ScrollNextRow

ScrollPriorPage

ScrollPriorRow
Second

Description
Obtains the number of seconds in the seconds portion of a time value.

Syntax
```plaintext
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
```plaintext
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:
```plaintext
SecondsAfter(21:15:30, 21:15:45)
```

This statement returns -15:
```plaintext
SecondsAfter(21:15:45, 21:15:30)
```

This statement returns 0:
```plaintext
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:

```powerscript
time start_time, end_time
start_time = 19:02:16
end_time = 19:02:28
```

then each of these statements returns 12:

```powerscript
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.

**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.

<table>
<thead>
<tr>
<th>To</th>
<th>Syntax 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move the read/write pointer in an OLE stream object.</td>
<td></td>
</tr>
<tr>
<td>Displays a specific frame in an AVI clip</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

**Description**

**Applies to**

OLEStream objects

**Syntax**

```powerscript
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>
</tbody>
</table>
**Seek**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>A long whose value is the position relative to origin to which you want to move the read/write pointer.</td>
</tr>
</tbody>
</table>
| origin (optional) | The value of the SeekType enumerated datatype specifying where you want to start the seek. Values are:  
  - FromBeginning! – (Default) At the beginning of the file  
  - FromCurrent! – At the current position  
  - FromEnd! – At the end of the file |

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

-1 Stream is not open  
-2 Seek error  
-9 Other error  

If any argument’s value is null, Seek returns null.

**Examples**  
This example writes additional data to an OLE stream. First, it opens an OLE object in the file MYSTUFF.OLE and assigns it to the OLEStorage object stg_stuff. Then it opens the stream called info in stg_stuff and assigns it to the stream object olestr_info. Seek positions the read/write pointer at the end of the stream so that the contents of the instance blob variable lb_info is written at the end.

The example 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  

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**

\[
\text{animationname}.\text{Seek}\ (s)
\]

**Argument** | **Description**  
--- | ---  
\[\text{animationname}\] | The name of animation control displaying the AVI clip  
\[s\] | A long value in the range 0 to 65,535 indicating the frame to display  

**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:

```power_script
// instance variable number
integer li_return

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**

\[
\text{rtename}.\text{SelectedColumn}\ ()
\]
SelectedIndex

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtename</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
global long ll_col, ll_line
global long ll_col = rte_Contact.SelectedColumn()
ll_line = rte_Contact.SelectedLine()
global 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
`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:

integer li_Index
li_Index = lb_actions. SelectedIndex()

These statements open the window w_emp if item 5 in lb_actions is selected:

integer li_X
li_X = lb_actions. SelectedIndex()
If li_X = 5 then Open(w_emp)

See also

SelectedItem

SelectedItem

Description

Obtains the text of the selected item in a ListBox control.

Applies to

ListBox and PictureListBox controls

Syntax

listboxname. SelectedItem ()

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.
SelectedLength

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 DropDownPictureListBox controls

Syntax

\[ \text{editname}.\text{SelectedLength}() \]

<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 DropDownPictureListBox control in which you want the length of the selected text. For a DataWindow, it reports the length of the selected text in the edit control over the current row and column.</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 DropDownListPictureListBox loses focus, the selected text is no longer selected.

Examples
If the selected text in the MultiLineEdit mle_Derivative is John Smith, then this example sets li_length to 10:

```PowerScript
long li_length
li_length = mle_Derivative.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.SelectedLine()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editname</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
SelectedPage

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:

```powerbuilder
integer li_SL
li_SL = mle_Contact.SelectedLine()
```

In this example, the line the user selects in the MultiLineEdit mle_winselect determines which window to open:

```powerbuilder
integer li_SL
li_SL = mle_winselect.SelectedLine()
IF li_SL = 1 THEN
  Open(w_emp_data)
ELSEIF li_SL = 2 THEN
  Open(w_dept_data)
END IF
```

See also

LineLength
Position
SelectedColumn
SelectedPage
SelectedText
TextLine

SelectedPage

Description

Obtains the number of the current page in a RichTextEdit control.

Applies to

RichTextEdit controls

Syntax

```powerbuilder
rtename.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 rtename. If an error occurs, SelectedPage returns -1.
**Usage**

The current page in a RichTextEdit control is the page that contains the insertion point in text entry mode or the page currently being displayed in preview mode.

When the RichTextEdit shares data with a DataWindow, SelectedPage returns the page number within the document instance for the current row.

For more information about document instances, see DataSource.

**Examples**

This example returns the page number of the current page:

```power
integer li_pagect
li_pagect = rte_1.SelectedPage()
```

**See also**

DataSource
PageCount
Preview
SelectedLength
SelectedLine
SelectedStart
SelectedText

---

**SelectedStart**

**Description**

Reports the position of the first selected character in an editable control.

**Applies to**

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, and DropDownPictureListBox controls

**Syntax**

```
editname.SelectedStart()
```

<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.
**SelectedText**

**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.

**Examples**

If the MultiLineEdit `mle_Comment` contains Closed for Vacation July 3 to July 10, and Vacation is selected, then this example sets `li_Start` to 12 (the position of the first character in Vacation):

```
integer li_Start
li_Start = mle_Comment.SelectedStart()
```

**See also**

Position
SelectedLine
SelectedPage

---

**SelectedText**

**Description**

Obtains the selected text in an editable control.

**Applies to**

DataWindow, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, RichTextEdit, DropDownListBox, and DropDownListListBox controls

**Syntax**

```
editname:SelectedText ( )
```

<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 DropDownListListBox control from which you want the selected text. For a DropDownListBox or DropDownListListBox, the AllowEdit property must be <code>true</code>. For a DataWindow, it reports the selected text in the edit control over the current row and column.</td>
</tr>
</tbody>
</table>
Return value  String. Returns the selected text in `editname`. If there is no selected text or if an error occurs, `SelectedText` returns the empty string (""). If `editname` is null, `SelectedText` returns null.

Usage  In a RichTextEdit control, any pictures in the selection are ignored. If the selection contains input fields, the names of the input fields, enclosed in brackets, become part of the string `SelectedText` returns. The contents of the input fields are not returned.

For example, when the salutation of a letter is selected, `SelectedText` might return:

```
Dear {title} {lastname}:
```

**Focus and the selection in a drop-down list**

When a DropDownListBox or DropDownPictureListBox loses focus, the selected text is no longer selected.

Examples  If the text in the MultiLineEdit `mle_Contact` is James B. Smith and James B. is selected, these statements set the value of `emp_fname` to James B:

```powerscript
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`:

```powerscript
string ls_Loc
ls_Loc = dddl_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
SelectedItem

Syntax

\[ \text{control}.\text{SelectionRange} \left( \text{startpos}, \text{endpos} \right) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>The name of the trackbar control</td>
</tr>
<tr>
<td>startpos</td>
<td>An integer that specifies the starting position of the range</td>
</tr>
<tr>
<td>endpos</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:

\[ \text{HTrackBar}.\text{SelectionRange} \left( 30, 70 \right) \]

See also

HTrackBar in PowerBuilder Objects and Controls
VTrackBar in PowerBuilder Objects and Controls

SelectedItem

Finds and highlights an item in a ListBox, DropDownListBox, or TreeView control.

To select an item | Use
-------------------|-------------------
In a ListBox control when you know the text of the item, but not its position | Syntax 1
In a ListBox control when you know the position of the item in the control’s list, or to clear the current selection | Syntax 2
In a TreeView control | Syntax 3

Syntax 1

When you know the text of an item

Description

Finds and highlights an item in a ListBox when you can specify some or all of the text of the item.

Applies to

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

Syntax

\[ \text{listboxname}.\text{SelectItem} \left( \text{item}, \text{index} \right) \]
### CHAPTER 10  PowerScript Functions

**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`.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listboxname</code></td>
<td>The name of the ListBox control in which you want to select a line</td>
</tr>
<tr>
<td><code>item</code></td>
<td>A string whose value is the starting text of the item you want to select</td>
</tr>
<tr>
<td><code>index</code></td>
<td>The number of the item after which you want to begin the search</td>
</tr>
</tbody>
</table>

**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:

```powerscript
integer li_Index
li_Index = lb_Actions.SelectItem("Delete Files", 2)
```

If item 4 in `lb_Actions` is Select Objects, this example starts searching after item 2, finds and highlights Select Objects, and sets `li_Index` to 4:

```powerscript
integer li_Index
li_Index = lb_Actions.SelectItem("Sel", 2)
```

**See also**
- `AddItem`
- `DeleteItem`
- `FindItem`
- `InsertItem`
- `SetState`


**Syntax 2**

**When you know the item number**

Finds and highlights an item in a ListBox when you can specify the index number of the item. You can also clear the selection by specifying zero as the index number.

**Applies to**

ListBox, DropDownListBox, PictureListBox, and DropDownPictureListBox controls

**Syntax**

`listboxname.SelectItem(itemnumber)`

<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:

```powerbuilder
integer li_Index
li_Index = lb_Actions.SelectItem(5)
```
This example clears the selection from the text box of the DropDownListBox

ddlb\_choices and sets \texttt{li\_Index} to 0:

\begin{verbatim}
   integer li\_Index
   li\_Index = ddlb\_choices.SelectItem(0)
\end{verbatim}

See also

AddItem
DeleteItem
FindItem
InsertItem
SetState

\section*{Syntax 3}
\subsection*{For TreeView controls}
\textbf{Description}
Selects a specified item.

\textbf{Applies to}
TreeView controls

\textbf{Syntax}
\texttt{treeviewname.SelectItem(itemhandle)}

\begin{tabular}{|l|l|}
\hline
Argument & Description \\
\hline
\texttt{treeviewname} & The name of the TreeView control in which you want to select an item \\
\hline
\texttt{itemhandle} & The handle of the specified item \\
\hline
\end{tabular}

\textbf{Return value}
Integer. Returns 1 if it succeeds and -1 if an error occurs.

\textbf{Usage}
Use the \texttt{FindItem} function to get handles for items at specific positions in the TreeView control.

\textbf{Examples}
This example selects the parent of the current TreeView item:

\begin{verbatim}
   long ll\_tvi, ll\_tparent
   int li\_tvret
   ll\_tvi = tv\_list.FindItem(CurrentTreeItem!, 0)
   ll\_tparent = tv\_list.FindItem(ParentTreeItem!, & ll\_tvi)
   li\_tvret = tv\_list.SelectItem(ll\_tparent)
\end{verbatim}

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**
```
opre olecontrol:SelectObject (selectstate)
```

<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 containing the object you want to select</td>
</tr>
<tr>
<td><code>selectstate</code></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`:
```
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><code>tabcontrolname</code></td>
<td>The name of the Tab control in which you want to select a tab</td>
</tr>
<tr>
<td><code>tabidentifier</code></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:

```powerShell
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`:

```powerShell
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:

```powerShell
userobject uo_tabpage
string ls_tabpage
ls_tabpage = "uo_fontsettings"
tab_1.OpenTab(uo_tabpage, ls_tabpage, 0)
tab_1.SelectTab(uo_tabpage)
```

See also: OpenTab

---

**SelectText**

Selects text in an editable control.

<table>
<thead>
<tr>
<th>To select text in</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any editable control, other than a RichTextEdit</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>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: 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
SelectText

Syntax

editname.SelectText ( start, length )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editname</td>
<td>The name of the DataWindow, EditMask, InkEdit, MultiLineEdit, SingleLineEdit, DropDownList Box, or DropDownPictureList Box control in which you want to select text.</td>
</tr>
<tr>
<td>start</td>
<td>A long specifying the position at which you want to start the selection.</td>
</tr>
<tr>
<td>length</td>
<td>A long specifying the number of characters you want to select. If length is 0, no text is selected but PowerBuilder moves the insertion point to the location specified in start.</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.

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

This statement sets the insertion point at the end of the text in the SingleLineEdit sle_name:

```
sle_name.SelectText (Len(sle_name.Text), 0)
```

This statement selects the entire contents of the SingleLineEdit sle_name:

```
sle_name.SelectText (1, Len(sle_name.Text))
```

The rest of these examples assume the MultiLineEdit mle_EmpAddress contains Boston Street.

The following statement selects the string ost and returns 3:

```
mle_EmpAddress.SelectText (2, 3)
```

The next statement selects the string oston Street and returns 12:

```
mle_EmpAddress.SelectText (2, &Len(mle_EmpAddress.Text))
```
These statements select the string *Bos*, returns 3, and sets the focus to mle_EmpAddress so that *Bos* is highlighted:

```powerscript
mle_EmpAddress.SelectText(1, 3)
mle_EmpAddress.SetFocus()
```

See also

- Len
- Position
- SelectedItem
- SelectedText
- SetFocus
- TextLine

### Syntax 2

**For RichTextEdit controls and presentation styles**

**Description**

Selects text beginning and ending at a line and character position in a RichTextEdit control.

**Applies to**

RichTextEdit and DataWindow controls

**Syntax**

```powerscript
rtename.SelectText( fromline, fromchar, tol ine, 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>
</tbody>
</table>
| `band` (optional) | A value of the Band enumerated datatype specifying the band in which to make the selection. Values are:
- Detail!
- Header!
- Footer!
  The default is the band that contains the insertion point. |

**Return value**

Long. Returns the number of characters selected. 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`. 
SelectText

Usage

The insertion point is at the "to" end of the selection, that is, the position specified by tolinc and tochar. If tolinc 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:

```plaintext
rte_1.SelectText(1,1, 3,4)
```

This statement sets the insertion point at the beginning of line 2:

```plaintext
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:

```plaintext
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:

```plaintext
long ll_length
//Make line 2 the current line
rte_1.SelectText(2,1, 0,0)

// Specify a position after the last character
ll_length = rte_1.LineLength() + 1

// Set the insertion point at the end
rte_1.SelectText(2,ll_length, 0,0)
rte_1.SetFocus()
```

This example selects the text from the insertion point to the end of the current line. If the current line is the last line, the reported line length is 1 greater than the number of character you can select, so the code adjusts for it:

```plaintext
long ll_insertline, ll_insertchar
long ll_line, ll_count
```
CHAPTER 10  PowerScript Functions

// Get the insertion point
rte_1.Position(ll_insertline, ll_insertchar)

// Get the line number and line length
ll_line = rte_1.SelectedLine()
ll_count = rte_1.LineLength()
// Line length includes the eof file character,
// which can't be selected
IF ll_line = rte_1.LineCount() THEN ll_count -= 1

// Select from the insertion point to the end of
// line
rte_1.SelectText(ll_insertline, ll_insertchar, &ll_line, ll_count)

See also
SelectedText
SelectTextAll
SelectTextLine
SelectTextWord

SelectTextAll

Description
Selects all the contents of a RichTextEdit control including any special
characters such as carriage return and end-of-file (EOF) markers.

Applies to
RichTextEdit and DataWindow controls

Syntax
rtnamename.SelectTextAll ( { band } )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| rtnamename | 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 | 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.
**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  
SelectTextLine  
SelectTextWord
SelectTextWord

Description
Selects the word containing the insertion point in a RichTextEdit control.

Applies to
RichTextEdit and DataWindow controls

Syntax
rtename.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:

```
  rte_1.SelectTextWord()
```

This example selects the word at the insertion point. If there is no word, it increments the position until it finds a word. It checks when it reaches the end of a line and wraps to the next line as it looks for a word. If this script is assigned to a command button and the button is clicked repeatedly, you step through the text word by word:

```
  integer li_rtn
  long llstart, lcstart, ll_lines, ll_chars

  ll_lines = rte_1.LineCount()
  ll_chars = rte_1.LineLength()

  li_rtn = rte_1.SelectTextWord()

  // -1 if a word is not found at the insertion point
  DO WHILE li_rtn = -1
```
SelectTextWord

// Get the position of the cursor
rte_1.Position(llstart, lcstart)

// Increment by 1 to look for next word
lcstart += 1
// If at end of line move to next line
IF lcstart >= ll_chars THEN
   lcstart = 1 // First character
   llstart += 1 // next line

   // If beyond last line, return
   IF llstart > ll_lines THEN
      RETURN 0
   END IF
END IF

// Set insertion point
rte_1.SelectText(llstart, lcstart, 0, 0)
// In case it's a new line, get new line length
// Can't do this until the ins pt is in the line
ll_chars = rte_1.LineLength()

// Select word, if any
li_rtn = rte_1.SelectTextWord()
LOOP

   // Add code here to process the word (for example,
   // passing the word to a spelling checker)
 See also

   SelectedText
   SelectText
   SelectTextAll
   SelectTextLine
Send

Description
Sends a message to a window so that it is executed immediately.

Syntax
Send ( handle, message#, lowword, long )

<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 SendMessage in Windows if it succeeds and -1 if an error occurs. If any argument’s value is null, Send returns null.

Usage
PowerBuilder’s Send function sends the message identified by message# and optionally, lowword and long, to the window identified by handle to the Windows function SendMessage. The message is sent directly to the object, bypassing the object’s message queue. Send waits until the message is processed and obtains the value returned by SendMessage.

Messages in Windows
Use the Handle function to get the Windows handle of a PowerBuilder object.

You specify Windows messages by number. They are documented in the file WINDOWS.H that is part of the Microsoft Windows Software Development Kit (SDK) and other Windows development tools.

Posting a message
Messages sent with Send are executed immediately. To post a message to the end of an object’s message queue, use the Post function.
This statement scrolls the window \texttt{w_emp} up one page:
\begin{verbatim}
Send(Handle(w_emp), 277, 2, 0)
\end{verbatim}

Both of the following statements click the CommandButton \texttt{cb_OK}:
\begin{verbatim}
Send(Handle(Parent), 273, 0, Handle(cb_OK))
\end{verbatim}
\begin{verbatim}
cb_OK.TriggerEvent(Clicked!)
\end{verbatim}

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:
\begin{verbatim}
Send(Handle(dw_whatever), 274, 61472, 0)
\end{verbatim}

This statement maximizes the DataWindow:
\begin{verbatim}
Send(Handle(dw_whatever), 274, 61488, 0)
\end{verbatim}

This statement returns the DataWindow to its normal, defined size:
\begin{verbatim}
Send(Handle(dw_whatever), 274, 61728, 0)
\end{verbatim}

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 \texttt{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 \texttt{Send}. In this example, idx is an integer instance variable for the window:
\begin{verbatim}
// Send the Windows message for LB_GETCURSEL
// to the list box
idx = Send(Handle(This), 1033, 0, 0)
idx = idx + 1
\end{verbatim}

See also
\begin{itemize}
  \item \texttt{Handle}
  \item \texttt{Post}
\end{itemize}
SeriesCount

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) }

Argument | Description
--- | ---
controlname | The name of the graph for which you want the number of series, or the name of the DataWindow control containing the graph.

graphcontrol (DataWindow control only) (optional) | A string whose value is the name of the graph in the DataWindow control for which you want the number of series.

Return value
Integer. Returns the number of series in the graph if it succeeds and -1 if an error occurs. 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 )

Argument | Description
--- | ---
controlname | The name of the graph in which you want the name of a series, 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 for which you want the name of a series
seriesnumber | The number of the series for which you want to obtain the name

Return value
String. Returns the name assigned to the series. If an error occurs, it returns the empty string ("""). If any argument’s value is null, SeriesName returns null.

Usage
Series are numbered consecutively, from 1 to the value returned by SeriesCount. When you delete a series, the series are renumbered to keep the numbering consecutive. You can use SeriesName to find out the name of the series associated with a series number.

Examples
These statements store in the variable ls_SeriesName the name of series 5 in the graph gr_product_data:

    string ls_SeriesName
    ls_SeriesName = gr_product_data.SeriesName(5)

These statements store in the variable ls_SeriesName the name of series 5 in the graph gr_computers in the DataWindow control dw_equipment:

    string ls_SeriesName
    ls_SeriesName = &
    dw_equipment.SeriesName("gr_computers", 5)

See also
CategoryName
DeleteSeries
FindSeries
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
\[ \text{oletxnobject}.\text{SetAbort}() \]

**Argument**

<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 `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:

```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()

IF /* test for client satisfaction */ THEN
    lotxn_obj.SetComplete()
ELSE
    lotxn_obj.SetAbort()
```

To roll back a transaction Use

For OLETxnObject objects Syntax 1
For TransactionServer objects Syntax 2
**SetAbort**

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**
`transactionserver.SetAbort( )`

<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 `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:

``````
// 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
SetComplete
Which

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

Sets the value in the specified argument element.

Applies to
Window ActiveX controls

Syntax
activexcontrol.SetArgElement(index, argument)

<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, this 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>
<tr>
<td>index</td>
<td>Integer specifying argument placement.</td>
</tr>
<tr>
<td>argument</td>
<td>Any specifying the argument value.</td>
</tr>
</tbody>
</table>

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:

```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);
    ...
}
```

See also
GetArgElement
GetLastReturn
InvokePBFunction
TriggerPBEvent
SetAutomationLocale

Description
Sets the language to be used in automation programming for an OLE object. Call SetAutomationLocale if you have programmed automation commands in a language other than the user’s locale.

Applies to
OLE objects

Syntax
olename.SetAutomationLocale (language, sortorder)

Argument | Description
---|---
olename | The name of the object for which you want to set the automation locale.
language | A value of the LanguageID enumerated 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:
- LanguageNeutral! – No language is assumed. Automation commands match the server’s default command set.
- LanguageUserDefault! – The language locale is taken from the user’s settings in the International control panel.
- LanguageSystemDefault! – The language locale is taken from the version of Windows that is installed on the user’s machine.

You can also specify a language or dialect, such as LanguagePolish! or LanguagePortuguese_Brazilian!
For the list of language-specific values for LanguageID, use the PowerBuilder Browser.

sortorder | A value of the LanguageSortID enumerated datatype specifying the sort order for the language. Values are:
- LanguageSortNative! – Use the traditional sort order of the selected language.
- LanguageSortUnicode! – Use the sort order defined for Unicode

Return value | Integer. Returns 0 if it succeeds and -1 if an error occurs.

Usage
For most situations, you 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!)
The system’s default locale (LanguageSystemDefault!)

5 English (LanguageEnglish!)

If PowerBuilder is successful in validating the name in any of the languages above, it resets the locale to the value that succeeded. While this may result in the wrong locale in ambiguous cases, it will probably simplify access to standard Microsoft Office products that ship with both localized and English function and property names.

If you specify a language with SetAutomationLocale, but the OLE server does not have function and property names in that language, your OLE automation commands will fail unless the above procedure finds a language that works. If you have called SetAutomationLocale, PowerBuilder’s procedure for finding the correct language can reset it, as described in the previous paragraph.

Examples

This example sets the language to German for an OLEObject called oleobj_report:

```powershell
oleobj_report.SetAutomationLocale(LanguageGerman!)
```

This example sets the language to German for an OLE control ole_1:

```powershell
ole_1.Object.SetAutomationLocale(LanguageGerman!)
```

---

## SetAutomationPointer

### Description

Sets the automation pointer of an OLEObject object to the value of the automation pointer of another object.

### Applies to

OLEObject

### Syntax

```powershell
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 <code>oleobject</code>.</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 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 descendent object to the value of the automation pointer in the `OLEObject` object. Then it sets a value in the worksheet using the descendent object, saves it to a different file, and destroys both objects:

```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**

```
oleobject.SetAutomationTimeout( interval )
```

<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 containing the object for which you want to set the timeout period.</td>
</tr>
<tr>
<td><code>interval</code></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 <code>interval</code> to the default value.</td>
</tr>
</tbody>
</table>
**SetAutomationTimeout**

**Return value**
Integer. Returns 0 if it succeeds and -1 if it fails.

**Usage**
This function passes the value of `interval` to PowerBuilder’s implementation of the IMessageFilter interface and determines how long PowerBuilder tries to complete an OLE procedure call. The value applies only when PowerBuilder is the OLE client, not when PowerBuilder is the OLE server.

**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):

```powershell
OLEObject ole1
integer rs
long interval

interval = 900000
ole1 = create OLEObject
rs = ole1.ConnectToObject("Excel.Application")
rs = ole1.SetAutomationTimeout(interval)
```
SetBoldDate

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: true – Set the date to bold 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: Once! – Set or clear the bold setting for the specified date (default) Monthly! – Using the day portion of the specified date, set or clear the bold setting for this day in all months 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:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
integer li_return
Date d
d = date("January 5, 2005")
li_return = mcVacation.SetBoldDate( d, false, Monthly!)
```

See also

ClearBoldDates

---

**SetByte**

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><code>blobvariable</code></td>
<td>A variable of the Blob datatype in which you want to insert a value of the Byte datatype</td>
</tr>
<tr>
<td><code>n</code></td>
<td>The number of the position in <code>blobvariable</code> at which you want to insert a value of the Byte datatype</td>
</tr>
<tr>
<td><code>b</code></td>
<td>Data of the Byte datatype that you want to set into <code>blobvariable</code> at position <code>n</code></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:

```
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>
<tr>
<td>alignment</td>
<td>A value of the Alignment enumerated datatype specifying how to align the column. Values are:</td>
</tr>
<tr>
<td></td>
<td>- Left! – Align the column at the left margin</td>
</tr>
<tr>
<td></td>
<td>- Right! – Align the column at the right margin</td>
</tr>
<tr>
<td></td>
<td>- Center! – Center the column between the left and right margins</td>
</tr>
<tr>
<td></td>
<td>- Justify! – Not valid for the SetColumn function</td>
</tr>
<tr>
<td>width</td>
<td>The width of the column for which you want to set column properties.</td>
</tr>
</tbody>
</table>
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:

```powerbuilder
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:

```powerbuilder
oletxnobject.SetComplete()
```

Argument | Description
---|---
`oletxnobject` | The name of the OLETxnObject variable that is connected to the COM object

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.
Examples

The following example shows the use of `SetComplete` 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()
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`.  

```powerscript
CHAPTER 10  PowerScript Functions

Examples

The following example shows the use of `SetComplete` 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()
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`.  

PowerScript Reference Volume 2

961
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:

```pascal
// 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`
CHAPTER 10  PowerScript Functions

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>
</tbody>
</table>
| `clipboardformat` | 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:  
  - ClipFormatBitmap!
  - ClipFormatDIB!
  - ClipFormatDIF!
  - ClipFormatEnhMetafile!
  - ClipFormatHdrop!
  - ClipFormatLocale!
  - ClipFormatMetafilePict!
  - ClipFormatOEMText!
  - ClipFormatPalette!
  - ClipFormatPenData!
  - ClipFormatRIFF!
  - ClipFormatSYLK!
  - ClipFormatText!
  - ClipFormatTIFF!
  - ClipFormatUnicodeText!
  - ClipFormatWave!  
If `clipboardformat` is an empty string or a null value, SetData transfers the data with the format ClipFormatText!. |
| `data`            | A string or blob whose value is the data you want to transfer. |

Return value

Integer. Returns 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 )

Argument | Description
--- | ---
string | The data you want to send to a DDE client application
applname (optional) | The DDE name for the client application
topic (optional) | A string whose value is the basic data grouping the DDE client application referenced
item (optional) | A string (data within topic)

Return value
Integer. Returns 1 if it succeeds. If an error occurs, SetDataDDE returns a negative integer. Values are:
-1 Function called in the wrong context
-2 Data not accepted

If any argument’s value is null, SetDataDDE returns null.

Usage
To enable DDE server mode in your PowerBuilder application, call the StartServerDDE function. Then DDE messages from a DDE client trigger events in the PowerBuilder window. It is up to you to decide how your application responds by writing code for those events. When an application requests data of the DDE server, it triggers a RemoteRequest event. You typically call SetDataDDE in the script for a window’s RemoteRequest event.
If a client application has established a hot link with a location in your PowerBuilder application, you can call `SetDataDDE` in an event for the object associated with the location. As a server application, you decide how location names map to the controls in your application. For example, your application can decide that the DDE name `loc1` refers to the SingleLineEdit `sle_name` and a client application can establish a hot link with "loc1." Then in the Modified event for `sle_name`, you can call `SetDataDDE` so that the client application receives changes each time `sle_name` is changed. Likewise, if `loc1` referred to a DataWindow, you can call `SetDataDDE` in the ItemChanged event for the DataWindow.

The `applname` argument refers to the client application that has established a channel or a hot link with your application. `Topic` and `item` refer to a topic and location recognized by your server application. You only need to specify these arguments to make it clear to the client application who should receive the message and what is being sent.

**Examples**

This statement illustrates how `SetDataDDE` is used in a script for a RemoteRequest event when another DDE application requests data. The data sent is the text of the SingleLineEdit `sle_Address`:

```
SetDataDDE(sle_Address.Text)
```

This statement illustrates how the optional arguments are specified:

```
SetDataDDE(sle_Address.Text, "MYDB", &
            "Employee", "Address")
```

See also

- `GetDataDDE`
- `StartServerDDE`

---

**SetDataPieExplode**

**Description**

Explodes a pie slice in a pie graph. The exploded slice is moved away from the center of the pie, which draws attention to the data. You can explode any number of slices of the pie.

**Applies to**

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

**Syntax**

```
controlname.SetDataPieExplode( { graphcontrol }, seriesnumber,
                              datapoint, percentage )
```
SetDataPieExplode

<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. (DataWindow control only) (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:

```
integer series, datapoint
gObjectType 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.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set the data point’s colors</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Set the line style and width for the data point</td>
<td>Syntax 2</td>
</tr>
<tr>
<td>Set the fill pattern or symbol for the data point</td>
<td>Syntax 3</td>
</tr>
</tbody>
</table>

### Syntax 1
**For setting a data point’s colors**

**Description**
Specifies the colors of a data point in a graph.

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

**Syntax**
```
controlname.SetDataStyle( { graphcontrol, } seriesnumber, datapointnumber, 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 data point, or 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 color of a data point. (optional)</td>
</tr>
<tr>
<td>seriesnumber</td>
<td>The number of the series in which you want to set the color of a data point.</td>
</tr>
<tr>
<td>datapointnumber</td>
<td>The number of the data point for which you want to set 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 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 colortype.</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.
**SetDataStyle**

**Usage**

To change the appearance of a series, use `SetSeriesStyle`. The settings you make for the series are the defaults for all data points in the series.

To reset the color of individual points back to the series color, call `ResetDataColors`.

For a graph in a DataWindow, you can specify the appearance of a data point in the graph before PowerBuilder draws the graph. To do so, define a user event for `pbm_dwngraphcreate` and call `SetDataStyle` in the script for that event. The event `pbm_dwngraphcreate` is triggered just before a graph is created in a DataWindow object.

**Examples**

This example checks the background color for data point 6 in the series named Salary in the graph `gr_emp_data`. If it is red, `SetDataStyle` sets it to black:

```plaintext
long color_nbr
integer SeriesNbr
// Get the number of the series
SeriesNbr = gr_emp_data.FindSeries("Salary")
// Get the background color
gr_emp_data.GetDataStyle(SeriesNbr, 6, & Background!, color_nbr)
// If color is red, change it to black
IF color_nbr = 255 THEN &
    gr_emp_data.SetDataStyle(SeriesNbr, 6, & Background!, 0)
```

These statements set the text (foreground) color to black for data point 6 in the series named Salary in the graph `gr_depts` in the DataWindow control `dw_employees`:

```plaintext
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> (optional)</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.</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>
<tr>
<td><code>linestyle</code></td>
<td>A value of the LineStyle enumerated datatype. Values are: Continuous! Dash! DashDot! DashDotDot! Dot! Transparent!</td>
</tr>
<tr>
<td><code>linewidth</code></td>
<td>An integer whose value is 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, `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 `pbn_dwngraphcreate` and call `SetDataStyle` in the script for that event. The event `pbn_dwngraphcreate` is triggered just before a graph is created in a DataWindow object.
SetDataStyle

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)
```

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>
</tbody>
</table>
### CHAPTER 10  PowerScript Functions

#### Function: SetDataStyle

- **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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>enumvalue</td>
<td>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!</td>
</tr>
</tbody>
</table>
SetDateLimits

Examples
This example checks the fill pattern used for data point 10 in the series named Costs in the graph gr_product_data. If it is diamond, then SetDataStyle changes it to solid:

```
integer SeriesNbr
FillPattern data_pattern

// Get the number of the series
SeriesNbr = gr_product_data.FindSeries("Costs")

// Get the current fill pattern
gr_product_data.GetDataStyle(SeriesNbr, 10, & data_pattern)

// If the pattern is diamond, change it to solid
IF data_pattern = Diamond! THEN &
   gr_product_data.SetDataStyle(SeriesNbr, &
                                  10, Solid!)
```

See also
GetDataStyle
GetSeriesStyle
SeriesName
SetSeriesStyle

SetDateLimits

Description
Sets the maximum and minimum date limits for the calendar.

Applies to
MonthCalendar control

Syntax
```
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:

```powerScript
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

**Syntax**  
`treeviewname.SetDropHighlight( 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 highlight an item as the target of a drag-and-drop operation</td>
</tr>
<tr>
<td><code>itemhandle</code></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.
**SetDynamicParm**

**Examples**

This example uses the TreeView Clicked event to set the current TreeView item as the drop target:

```plaintext
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**

```plaintext
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.

**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:

```plaintext
SQLDA.SetDynamicParm(1, "MA")
```
This statement fills the fourth input parameter descriptor in SQLDA with the number 01742:

```powerscript
SQLDA.SetDynamicParm(4, "01742")
```

This statement fills the third input parameter descriptor in SQLDA with the date 12-31-2002:

```powerscript
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**
```
treeviewname.SetFirstVisible ( 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 identify an item as the first visible item</td>
</tr>
<tr>
<td>itemhandle</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:

```powerscript
long ll_tvi
int li_tvret
```
SetFocus

Description
Sets the focus on the specified object or control.

Applies to
Any object

Syntax
`objectname.SetFocus()`

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If `objectname` is null, `SetFocus` returns null.

Usage
If `objectname` is a ListBox, `SetFocus` displays the focus rectangle around the first item. If `objectname` is a DropDownListBox, `SetFocus` highlights the edit box. To select an item in a ListBox or DropDownListBox, use `SelectItem`.

Drawing objects cannot have focus. Therefore, you cannot use `SetFocus` to set focus to in a Line, Oval, Rectangle, or RoundRectangle.

Examples
This statement in the script for the Open event in a window moves the focus to the first item in `lb_Actions`:

```
lb_Actions.SetFocus()
```

See also
SetItem
SetState
SetTop
**SetGlobalProperty**

**Description**
Sets the value of an SSL global property.

**Applies to**
SSLServiceProvider object

**Syntax**
```
sslserviceprovider.SetGlobalProperty (property, value)
```

**Argument** | **Description**
--- | ---
`sslserviceprovider` | Reference to the SSLServiceProvider service instance.
`property` | The name of the SSL property you want to set. For a complete list of supported SSL properties, see your EAServer documentation or the online Help for the Connection object.
`value` | String value of the SSL property.

**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**

Sets data associated with a ListView item to the property values you specify in a ListViewItem variable.

**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 )
```

**Argument**

<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 ListView item:

```powerscript
listviewitem lvi_1
lvi_1.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**

```
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.
This example assigns display values to three columns in a report view for three `lv_list` ListView items:

```powerbuilder
listviewitem l_lvi
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")
```

See also
- `GetItem`

### Syntax 3

#### For TreeView controls

Sets the data associated with a specified item.

- **Applies to**
  - TreeView controls

- **Syntax**
  ```powerbuilder
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>
<tr>
<td>itemhandle</td>
<td>The handle associated with the item you want to change</td>
</tr>
<tr>
<td>item</td>
<td>The TreeView item you want to change</td>
</tr>
</tbody>
</table>

- **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:

```powerlanguage
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`

---

**SetLevelPictures**

Description

Sets the picture indexes for all items at a particular level.

Applies to

TreeView controls

Syntax

```powerlanguage
treeviewname.SetLevelPictures( level, pictureindex, selectedpictureindex, statepictureindex, overlaypictureindex)
```

<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 set the pictures for a given TreeView level</td>
</tr>
<tr>
<td>level</td>
<td>The TreeView level for which you are setting the picture indexes</td>
</tr>
<tr>
<td>pictureindex</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>selectedpictureindex</td>
<td>An index from the regular picture list specifying the picture to be displayed when the item is selected</td>
</tr>
<tr>
<td>statepictureindex</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>overlaypictureindex</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>

Return value

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

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:

```c
long ll_tvi, ll_child, ll_child2
int li_pict, li_level
treewitem l_tvi

li_level = 6
tv_list.SetLevelPictures( 3, li_level, li_level, &
    li_level, li_level)

ll_tvi = tv_list.FindItem(RootTreeItem! , 0)
ll_child = tv_list.InsertItemLast(ll_tvi, "Walton",2)
ll_child2 = tv_list.InsertItemLast(ll_child, &
    "Spitfire Suite", li_level)
tv_list.ExpandItem(ll_child)
tv_list.SetFirstVisible(ll_child)
```

See also
AddPicture

SetLibraryList

Description
Changes the files in the library search path of the application 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 )
```

<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>

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.
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:

```powerbuilder
string ls_list

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 )

Argument | Description
--- | ---
editmaskname | The name of the EditMask for which you want to specify the edit mask.
maskdatatype | A MaskDataType enumerated datatype indicating the datatype of the mask. Values are:
- DateMask!
- DateTimeMask!
- DecimalMask!
- NumericMask!
- StringMask!
- TimeMask!
mask | A string whose value is the edit mask.

Return value
Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, SetMask returns null.

Usage
In an edit mask, a fixed set of characters represent a type of character that the user can enter. In addition, punctuation controls the format of the entered value. Each mask 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 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:

```power
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:

```power
password_mask.SetMask(NumericMask!, "#####")
```

**SetMessage**

Description

Sets an error message for an object of type Throwable.

Syntax

`throwableobject.SetMessage (newMessage )`

<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:

```power
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:

```power
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

SetMicroHelp

Description
Specifies the text to be displayed in the MicroHelp box in an MDI frame window.

Applies to
MDI frame windows

Syntax
\texttt{windowname.SetMicroHelp ( string )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowname</td>
<td>The name of the MDI frame window with MicroHelp for which you want to set the MicroHelp text</td>
</tr>
<tr>
<td>string</td>
<td>A string whose value is the new MicroHelp text</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:

```powerscript
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:

```powerscript
W_New.SetMicroHelp(This.Tag)
```

---

### SetNull

**Description**

Sets a variable to `null`. The variable can be any datatype except for an array, structure, or autoinstantiated object.

**Syntax**

```
SetNull ( anyvariable )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>anyvariable</code></td>
<td>The variable you want to set to null</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`, `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`:

```powerscript
SetNull(Salary)
```

**See also**

`IsNull`
SetOverlayPicture

Description
Puts an image in the control’s image list into an overlay image list.

Applies to
ListView and TreeView controls

Syntax
controlname.SetOverlayPicture (overlayindex, imageindex)

Argument Description
controlname The name of the ListView or TreeView control to which you want to add an overlay image.
overlayindex The index number of the overlay picture in the overlay image list. The overlay image list is a 1-based array. Overlayindex must be 1 (for the first image), a previously designated index (replacing an image), or 1 greater than the current largest index (adding another image). SetOverlayPicture fails if you specify an index that creates gaps in the array.
imageindex The index number of an image in the control’s main image list. For ListViews, both the large and small pictures at that index become overlay images. The image is still available for use as an item’s main image.

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

Usage
An overlay picture must have the same height and width as the picture it is used to overlay. The color specified in the SetPictureMask property when the picture is inserted becomes transparent when the picture is used as an overlay, allowing part of the original image to be visible beneath the overlay.

The overlay list acts as a pointer back to the source image in the regular picture lists. If you delete an image that is also used in the overlay list, the displayed overlay pictures are affected too.

Examples
This example designates overlay images in a ListView control. The same picture is used for large and small images:

// Set up the overlay images
integer index
index = lv_1.AddLargePicture("shortcut.ico")
index = lv_1.AddSmallPicture("shortcut.ico")
lv_1.SetOverlayPicture(1, index)
index = lv_1.AddLargePicture("not.ico")
index = lv_1.AddSmallPicture("not.ico")
lv_1.SetOverlayPicture(2, index)
// Assign the second overlay image to the first item
listviewitem lvi
integer i
i = lv_1.GetItem(1, lvi)
lvi.OverlayPictureIndex = 2
i = lv_1.SetItem(1, lvi)

This example designates the first picture in the TreeView’s main image list as the first overlay picture. The picture was added to the main image list on the TreeView’s property sheet:

tv_list.SetOverlayPicture(1, 1)

This code in the TreeView’s Clicked event assigns the overlay image to the clicked item:

treeviewitem tvi
tv_list.GetItem(handle, tvi)
tvi.OverlayPictureIndex = 1
tv_list.SetItem(handle, tvi)
SetParagraphSetting

**Description**
Sets the size of the indentation, left margin, or right margin of the paragraph containing the insertion point in a RichTextEdit control.

**Applies to**
RichTextEdit controls

**Syntax**

```
rtecontrol.SetParagraphSetting ( whichsetting, value )
```

**Argument** | **Description**
--- | ---
*rtecontrol* | The name of the control for which you want paragraph information.
*whichsetting* | A value of the ParagraphSetting enumerated 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.

**Usage**
Each paragraph has indentation, left margin, and right margin settings. To set all three for the current paragraph, call SetParagraphSetting three times.

**Examples**
This example sets the indentation setting for the current paragraph to a quarter inch:

```
ll_indent = rte_1.SetParagraphSetting(Indent!, 250)
```

This example sets the left margin for the current paragraph to an inch:

```
rte_1.SetParagraphSetting(LeftMargin!, 1000)
```

**See also**
GetParagraphSetting
SetAlignment
SetSpacing
SetTextColor
SetTextStyle
**SetPicture**

**Description**
Assigns an image stored in a blob to be the image in a Picture control.

**Applies to**
Picture controls

**Syntax**
```
picturecontrol.SetPicture ( bimage )
```

<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>
<tr>
<td>bimage</td>
<td>A blob containing the new bitmap. <code>bimage</code> 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:

```powerShell
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
```

---

*CHAPTER 10  PowerScript Functions*
SetPointer

Sets the mouse pointer to the specified shape.

Syntax

```
SetPointer ( type )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A value of the Pointer enumerated datatype indicating the type of pointer you want. Values are: Arrow! Cross! Beam! HourGlass! SizeNS! SizeNESW! SizeWE! SizeNWSE! UpArrow!</td>
</tr>
</tbody>
</table>

Return value

Pointer. Returns the enumerated type of the pointer it replaced so the script can restore it, if necessary. If type is null, SetPointer returns null.

Usage

Use SetPointer to display an hourglass at the beginning of a script when the script will take a long time to execute. The pointer remains set until you change it again in the script or the script terminates.

Restoring the arrow pointer

The pointer automatically changes back to an arrow when the script finishes executing. You do not have to change it back to an arrow.

In PowerBuilder’s painters, you can specify the pointer shape that PowerBuilder displays when the user moves the pointer over a window, a control, or specific parts of a DataWindow object. The available shapes include the stock pointers listed above, as well as any custom cursor files you have.

Examples

This statement sets the pointer to the hourglass shape:

```
SetPointer(HourGlass!)
```

This example saves the old pointer and restores it when a long activity is completed:

```
pointer oldpointer // Declares a pointer variable
oldpointer = SetPointer(HourGlass!)
... // Performs some long activity
SetPointer(oldpointer)
```
SetPosition

Specifies the front-to-back position of a control in a window, a window, or an object within a DataWindow.

<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. <em>ObjectName</em> cannot be a child window or a sheet.</td>
</tr>
<tr>
<td><code>position</code></td>
<td>A SetPosType enumerated datatype. The values you can specify depend on whether <code>objectname</code> is a control or a window.</td>
</tr>
<tr>
<td></td>
<td>For controls, values are:</td>
</tr>
<tr>
<td></td>
<td>• Behind! – Position <code>objectname</code> behind <code>precedingobject</code> in the order</td>
</tr>
<tr>
<td></td>
<td>• ToTop! – Position <code>objectname</code> on top of all other controls</td>
</tr>
<tr>
<td></td>
<td>• ToBottom! – Position <code>objectname</code> behind all other controls</td>
</tr>
<tr>
<td></td>
<td>For windows, values are:</td>
</tr>
<tr>
<td></td>
<td>• TopMost! – Always display <code>objectname</code> on top of all other open windows</td>
</tr>
<tr>
<td></td>
<td>• NoTopMost! – Do not always display <code>objectname</code> on top of all other open windows</td>
</tr>
<tr>
<td><code>precedingobject</code></td>
<td>The name of the object you want to position <code>objectname</code> behind. <code>Precedingobject</code> is required if <code>position</code> is Behind!.</td>
</tr>
</tbody>
</table>
**SetPosition**

**Return value**
Integer. Returns 1 when it succeeds and -1 if an error occurs. If any argument’s value is null, SetPosition returns null.

**Usage**
The front-to-back order for controls determines which control covers another when they overlap. If a control completely covers another control, the control that is in back becomes inaccessible to the user.

When you specify TopMost! for more than one window, the most recently executed SetPosition function controls which window displays on top.

**Examples**
This statement positions cb_two on top:
```
cb_two.SetPosition(ToTop!)
```

This statement positions cb_two behind cb_three:
```
cb_two.SetPosition(Behind!, cb_three)
```

This statement makes the window w_signon the topmost window:
```
w_signon.SetPosition(TopMost!)
```

This statement makes the window w_signon no longer necessarily the topmost window:
```
w_signon.SetPosition(NoTopMost!)
```

**Syntax 2**

**For positioning objects within a DataWindow**
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>
</tbody>
</table>
Return value

Integer. Returns 1 when it succeeds and -1 if an error occurs. If any argument’s value is null, SetPosition returns null.

Examples

This statement moves oval_red in dw_rpt to the header and brings it to the front:

```plaintext
dw_rpt.setPosition("oval_red", "header", TRUE)
```

This statement does not change the position of oval_red, but does bring it to the front:

```plaintext
dw_rpt.setPosition("oval_red", ",", TRUE)
```

This statement moves oval_red to the footer but does not bring it to the front:

```plaintext
dw_rpt.setPosition("oval_red", "footer", FALSE)
```

**SetProfileString**

**Description**

Writes a value in a profile file for a PowerBuilder application.

**Syntax**

```plaintext
SetProfileString ( filename, section, key, value )
```

<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 include the full path in filename, PowerBuilder searches the DOS path for filename.</td>
</tr>
<tr>
<td>section</td>
<td>A string whose value is the name of a group of related values in the profile file. If section does not exist in the file, PowerBuilder adds it.</td>
</tr>
</tbody>
</table>
SetProfileString

<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 section for which you want to specify a value. If key does not exist in section, PowerBuilder adds it.</td>
</tr>
<tr>
<td>value</td>
<td>A string whose value is the value you want to specify for key.</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.

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.

Examples

This statement sets the keyword Title in section Position of file C:\PROFILE.INI to the string MGR:

```powerscript
SetProfileString("C:\PROFILE.INI", & "Position", "Title", "MGR")
```

See also

ProfileInt
ProfileString

---

**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 and trackbar controls

Syntax

```powerscript
controlname.SetRange ( startpos, endpos )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>controlname</td>
<td>The name of the progress bar or trackbar</td>
</tr>
<tr>
<td>startpos</td>
<td>Integer indicating the initial position of the range</td>
</tr>
<tr>
<td>endpos</td>
<td>Integer indicating the terminal position of the range</td>
</tr>
</tbody>
</table>

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:

```powerscript
HProgressBar.SetRange ( 1, 10 )
```

See also

OffsetPos
SelectionRange
StepIt
SetRecordSet

SetRecordSet

Sets an ADOResultSet object to obtain its data and metadata from a passed
ADO Recordset.

Applies to
ADOResultSet objects

Syntax
adoreresultset.SetRecordSet ( adoredcobject )

Argument | Description
--- | ---
adoreresultset | An ADOResultSet object into which the function places the
passed ADO Recordset.
adoredcobject | An OLEObject object that contains an ADO Recordset.
Passing an OLEObject that does not contain an ADO Recordset generates an error.

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:

```
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)
```
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`:

```powershell
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:

```powershell
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.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make a single DDE request of a server application (a cold link)</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Make a DDE request of a server application when you have established a warm link by opening a channel</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>

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 )

Argument | Description
--- | ---
location | A string whose value is the location of the data in the server application that will accept the data. The format of location depends on the application that will receive the request.
value | A string whose value you want to send to the remote application.
applname | A string whose value is the DDE name of the server application.
topicname | A string identifying the data or the instance of the application that will accept the data (for example, in Microsoft Excel, the topic name could be the name of an open spreadsheet).

Return value

Integer. Returns 1 if it succeeds and a negative integer if an error occurs. Values are:

-1 Link was not started
-2 Request denied

If any argument’s value is null, SetRemote returns null.

Usage

When using DDE, your PowerBuilder application must have an open window, which will be the client window. For this syntax, the active window is the DDE client window.

For more information about DDE channels and warm and cold links, see the ExecRemote function.
Examples
This statement asks Microsoft Excel to set the value of the data in row 5, column 7 of a worksheet called SALES.XLS to 4500:

SetRemote("R5C7", "4500", "Excel", "SALES.XLS")

See also
ExecRemote
GetRemote
OpenChannel

Syntax 2
For DDE requests via an open channel

Description
Asks a DDE server application to accept data to be stored in the specified location when you have already established a warm link by opening a channel to the server. A warm link, with an open channel, is more efficient when you intend to make several DDE requests.

Syntax
SetRemote ( location, value, handle {, windowhandle } )

<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 (optional)</td>
<td>The handle to the window that is acting as the DDE client.</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.
**SetResultSet**

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:

```powerbuilder
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**: `adoresultset.SetResultSet(resultsetobject)`
- **Argument Description**:
  - `adoresultset`: An ADOResultSet object into which the function places the passed result set as an ADO Recordset
  - `resultsetobject`: A ResultSet object that contains result set data
- **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`.

**See also**
- `GenerateResultSet` method for DataWindows in the *DataWindow Reference* or the online Help
- `GetRecordSet`
- `SetRecordSet`
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 )`

Argument | Description
---|---
controlname | The name of the graph in which you want to set the color of a series, or the name of the DataWindow control containing the graph.
graphcontrol | 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)
seriesname | A string whose value is the name of the series for which you want to set the color.
colortype | A value of the grColorType enumerated 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)
color | A long specifying the new color for colortype.

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If any argument’s value is null, SetSeriesStyle returns null.
Usage

Data points in a series can have their own style settings. Settings made with `SetDataStyle` set the style of individual data points and override series settings.

The graph stores style information for properties that 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 text (foreground) color of the series named `Salary` in the graph `gr_emp_data` to black:

```powerbuilder
gr_emp_data.SetSeriesStyle("Salary", &
   Foreground!, 0)
```

This statement sets the background color of the series named `Salary` in the graph `gr_depts` in the `DataWindow` control `dw_employees` to black:

```powerbuilder
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:

```powerbuilder
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_sales_data.SetSeriesStyle("northeast", &
      line_color, Line!, 0)
```
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
controlname.SetSeriesStyle ({ 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 in which you want to set the line style and width of a 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 in which you want to set the line style and width. (DataWindow control only)</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
</tr>
<tr>
<td>seriesname</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: Continuous! Dash! DashDot! DashDotDot! Dot! 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.
SetSeriesStyle

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:

```
gr_product_data.SetSeriesStyle("Costs", & Dot!, 5)
```

See also
GetDataStyle
GetSeriesStyle
SeriesName
SetDataStyle

Syntax 3
For the fill pattern and symbols in a graph

Description
Specifies the fill pattern and symbol for data markers in a series.

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

Syntax
```
controlname.SetSeriesStyle( graphcontrol, seriesname, 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 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 in which you want to set the appearance.</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
</tr>
<tr>
<td>seriesname</td>
<td>A string whose value is the name of the series in which you want to set the appearance.</td>
</tr>
</tbody>
</table>
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, 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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enumvalue</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>
<tr>
<td>To change the fill pattern, use a FillPattern value:</td>
<td></td>
</tr>
<tr>
<td>Bdiagonal! (Lines from lower left to upper right)</td>
<td></td>
</tr>
<tr>
<td>Diamond!</td>
<td></td>
</tr>
<tr>
<td>Fdiagonal! (Lines from upper left to lower right)</td>
<td></td>
</tr>
<tr>
<td>Horizontal!</td>
<td></td>
</tr>
<tr>
<td>Solid!</td>
<td></td>
</tr>
<tr>
<td>Square!</td>
<td>Vertical!</td>
</tr>
<tr>
<td>To change the symbol type, use a grSymbolType value:</td>
<td></td>
</tr>
<tr>
<td>NoSymbol!</td>
<td></td>
</tr>
<tr>
<td>SymbolHollowBox!</td>
<td></td>
</tr>
<tr>
<td>SymbolX!</td>
<td></td>
</tr>
<tr>
<td>SymbolStar!</td>
<td></td>
</tr>
<tr>
<td>SymbolHollowUpArrow!</td>
<td></td>
</tr>
<tr>
<td>SymbolHollowCircle!</td>
<td></td>
</tr>
<tr>
<td>SymbolHollowDiamond!</td>
<td></td>
</tr>
<tr>
<td>SymbolSolidDownArrow!</td>
<td></td>
</tr>
<tr>
<td>SymbolSolidUpArrow!</td>
<td></td>
</tr>
<tr>
<td>SymbolSolidCircle!</td>
<td></td>
</tr>
<tr>
<td>SymbolSolidDiamond!</td>
<td></td>
</tr>
<tr>
<td>SymbolPlus!</td>
<td></td>
</tr>
<tr>
<td>SymbolHollowDownArrow!</td>
<td></td>
</tr>
<tr>
<td>SymbolSolidBox!</td>
<td></td>
</tr>
</tbody>
</table>

**Argument**

enumvalue

**Description**

A value of an enumerated datatype specifying an appearance setting for the series. Values for the FillPattern or grSymbolType enumerated datatypes follow.
SetSeriesStyle

Examples

This statement sets the symbol used for the series named Costs in the graph gr_product_data to a plus sign:

```powerbuilder
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:

```powerbuilder
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**

```powerbuilder
controlname.SetSeriesStyle ( { graphcontrol, } seriesname, overlaystyle )
```

<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 overlay status of a 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 in which you want to set the overlay status. (DataWindow control only) (optional)</td>
</tr>
<tr>
<td>seriesname</td>
<td>A string whose value is the name of the series whose overlay status you want to change.</td>
</tr>
<tr>
<td>overlaystyle</td>
<td>A boolean value indicating whether you want the series to be an overlay, meaning that the series is shown in front as a line. Set overlaystyle to true to make the specified series an overlay. Set it to false to remove the overlay setting.</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

For a graph in a DataWindow, you can specify the appearance of a series in the graph before PowerBuilder draws the graph. To do so, define a user event for 

`pbm_dwngraphcreate` and call `SetSeriesStyle` in the script for that event. The event `pbm_dwngraphcreate` is triggered just before a graph is created in a DataWindow object.

Examples

This statement sets the style of the series named `Costs` in the graph `gr_product_data` to overlay:

```
gr_product_data.SetSeriesStyle("Costs", TRUE)
```

These statements in the Clicked event of the DataWindow control `dw_employees` store the style of the series under the pointer in the graph `gr_depts` in the variable `style_type`. If the style of the series is overlay (true), the script changes the style to normal (false):

```
string SeriesName
integer SeriesNbr, Data_Point
boolean overlay_style
grObjectType MouseHit

MouseHit = dw_employees.ObjectAtPointer( &
    "gr_depts", SeriesNbr, Data_Point)

IF MouseHit = TypeSeries! THEN
    SeriesName = &
    dw_employees.SeriesName("gr_depts",SeriesNbr)
    dw_employees.GetSeriesStyle("gr_depts", &
        SeriesName, overlay_style)

    IF overlay_style THEN &
        dw_employees.SetSeriesStyle("gr_depts", &
            SeriesName, FALSE)
    END IF

See also

GetDataStyle
GetSeriesStyle
SeriesName
SetDataStyle
**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><code>controlname</code></td>
<td>The name of the MonthCalendar control for which you want to set the selected date</td>
</tr>
<tr>
<td><code>d</code></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`.

Examples

This example sets the selected date to a date passed into a function:

```
// function argument seldate
integer li_return

li_return = mc_1.SetSelectedDate(seldate)
```

See also

GetSelectedDate

SetDateLimits
SetSelectedRange

Description
Sets the range of selected dates.

Applies to
MonthCalendar control

Syntax
controlname.SetSelectedRange ( start, end )

Argument | Description
--- | ---
controlname | The name of the MonthCalendar control for which you want to set the selected range
start | A date value to be set as the earliest date selected
end | A date value to be set as the latest date selected

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.

Examples
This example sets the start date of the selected range to startdate and the end date to enddate:

```power
integer li_return
Date startdate, enddate

startdate = Today()
enddate = Date("12-31-2007")
li_return = mc_1.SetSelectedRange(startdate, enddate)
```

See also
GetSelectedRange
SetDateLimits
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

\texttt{rtename.SetSpacing ( \textit{spacing} )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{rtename}</td>
<td>The name of the RichTextEdit control in which you want to set the line spacing.</td>
</tr>
<tr>
<td>\textit{spacing}</td>
<td>A value of the Spacing enumerated datatype specifying the line spacing for the text. Values are:</td>
</tr>
<tr>
<td></td>
<td>Spacing1! – Single spacing</td>
</tr>
<tr>
<td></td>
<td>Spacing15! – One and a half line spacing</td>
</tr>
<tr>
<td></td>
<td>Spacing2! – Double spacing</td>
</tr>
</tbody>
</table>

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 \texttt{rte_1}:

\begin{verbatim}
rte_1.SetSpacing(Spacing2!)
\end{verbatim}

This example specifies one and a half line spacing:

\begin{verbatim}
rte_1.SetSpacing(Spacing15!)
\end{verbatim}

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 of 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>
<tr>
<td>state</td>
<td>A boolean value that determines the state of the item:</td>
</tr>
<tr>
<td></td>
<td>• TRUE – Selected</td>
</tr>
<tr>
<td></td>
<td>• FALSE – Not selected</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, SetState returns null.

Usage

When the MultiSelect property for the control is false, use SelectItem, instead of SetState, to select one item at a time.

Examples

This statement turns on the highlight for item 6 in `lb_Actions`:

```
lb_Actions.SetState(6, TRUE)
```

This statement deselects all items in `lb_Actions`:

```
lb_Actions.SetState(0, FALSE)
```

This statement turns off the highlight for item 6 in `lb_Actions` if it is selected and turns it on again if it is not selected:

```
IF lb_Actions.State(6) = 1 THEN
   lb_Actions.SetState(6, FALSE)
ELSE
   lb_Actions.SetState(6, TRUE)
END IF
```

See also

SelectItem
SetTop
State
SetTextColor

Description
Sets the color of selected text in a RichTextEdit control.

Applies to
RichTextEdit controls

Syntax
```plaintext
rtename.SetTextColor( colornumber )
```

<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 color of selected text.</td>
</tr>
<tr>
<td>colornumber</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:
```plaintext
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
```plaintext
rtename.SetTextStyle( bold, underline, {subscript}, {superscript}, italic, strikeout )
```

<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 specify formatting for selected text.</td>
</tr>
<tr>
<td>bold</td>
<td>A boolean value specifying whether the selected text is bold.</td>
</tr>
<tr>
<td>underline</td>
<td>A boolean value specifying whether the selected text is underlined.</td>
</tr>
<tr>
<td>subscript</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>(obsolete)</td>
<td></td>
</tr>
</tbody>
</table>

Printed Page: 1014
CHAPTER 10  PowerScript Functions

Return value

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

Examples

This example makes selected text in the RichTextEdit `rte_1` bold and italic:

```
rte_1.SetTextStyle(TRUE, FALSE, &
TRUE, FALSE)
```

This example makes the selected text italic but keeps other text formatting as it was:

```
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 )
```

Return value

Boolean. Returns true if it succeeds and false if an error occurs.

<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>seconds</code></td>
<td>An unsignedlong that specifies the number of seconds that elapse before a transaction is rolled back</td>
</tr>
</tbody>
</table>
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. EA Server 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:

```powershell
// 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
\texttt{controlname.SetToday(\textit{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>\textit{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:

\begin{verbatim}
Date currentdate
integer li_return

  currentdate = Date(em_1.Text)
  li_return = mc_1.SetToday(currentdate)
\end{verbatim}

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 } } )
```

### Argument Description

<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>
</tbody>
</table>
| `visible`  | A boolean value specifying whether to make the toolbar visible. Values are:  
- TRUE – Make the toolbar visible  
- FALSE – Hide the toolbar |
| `alignment` | (optional) A value of the ToolbarAlignment enumerated datatype specifying the alignment for the toolbar. Values are:  
- AlignAtTop! – Dock the toolbar at the top of the frame.  
- AlignAtLeft! – Dock the toolbar on the left side of the frame.  
- AlignAtRight! – Dock the toolbar on the right side of the frame.  
- AlignAtBottom! – Dock the toolbar at the bottom of the frame.  
- Floating! – Float the toolbar. The floating toolbar has its own frame and miniature title bar |
| `floatingtitle` | (optional) A string whose value is the title for the toolbar when its alignment is Floating!. |

### Return value
Integer. Returns 1 if it succeeds. `SetToolbar` returns -1 if there is no toolbar for the index you specify or if an error occurs. If any argument’s value is null, returns null.

### Usage
When you use `SetToolbar` to change the toolbar alignment from a docked position to Floating!, PowerBuilder uses the last known position information unless you also call `SetToolbarPos` to adjust the position.

The toolbars are not redrawn until the script ends, so setting the alignment with `SetToolbar` and the position with `SetToolbarPos` looks like a single change to the user.
Examples

This example allows the user to choose an alignment in a ListBox `lb_position`. The selected string is converted to a ToolbarAlignment enumerated value, which is used to change the alignment of toolbar index 1:

```powerscript
toolbaralignment tba_align

CHOSE 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:

```powerscript
Parent.i_toolbaralign = AlignAtTop!
```

This script changes the toolbar alignment:

```powerscript
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.

Applies to MDI frame and sheet windows

\[ \text{window.SetToolbarPos( toolbarindex, dockrow, offset, insert )} \]

<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>dockrow</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>offset</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, ( \text{offset} ) is measured from the left edge. For toolbars on the left or right, ( \text{offset} ) is measured from the top.</td>
</tr>
<tr>
<td></td>
<td>If ( \text{insert} ) is true, the ( \text{offset} ) you specify is adjusted so that the toolbar does not overlap others in the row.</td>
</tr>
<tr>
<td></td>
<td>Specify an offset of 0 to position the toolbar ahead of other toolbars in ( \text{dockrow} ).</td>
</tr>
<tr>
<td>insert</td>
<td>A boolean value specifying whether you want to insert the specified toolbar before the toolbars in ( \text{dockrow} ) causing them to move over or down a row, or you want to add ( \text{toolbarindex} ) to ( \text{dockrow} ). Values are:</td>
</tr>
<tr>
<td></td>
<td>• TRUE – Move any toolbars already in ( \text{dockrow} ) 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 ( \text{dockrow} ). Its position in relation to other toolbars in the row is determined by ( \text{offset} ).</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, returns null.

Usage

To find out whether the docked toolbar is at the top, bottom, left, or right edge of the window, call GetToolbar.

If the toolbar’s alignment is floating, instead of docked, then values you specify with Syntax 1 of SetToolbarPos take effect when you change the alignment to a docked position with SetToolbar.

When insert is false, to move the toolbar before other toolbars in dockrow, specify a value that is less than the offset for the existing toolbars. If there is already a toolbar at offset 1, then you can move the toolbar to the beginning of the row by setting offset to 0. If offset is equal to or greater than the offset of existing toolbars, but less than their end, the newly positioned toolbar will begin just after the existing one. Otherwise, the toolbar will be positioned at offset.

If the user drags the toolbar to a docked position, the new row and offset replace values set with SetToolbarPos.

Examples

This example docks toolbar 1 at the left, adding it to docking row 1:

```power_script
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:

```power_script
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:

```power_script
w_frame.SetToolbar(1, TRUE, AlignAtLeft!)
w_frame.SetToolbarPos(1, 2, 1, TRUE)
```

See also

GetToolbar
GetToolbarPos
SetToolbar
**SetToolbarPos**

### Syntax 2

**For floating toolbars**

Sets the position and size of a floating toolbar.

**Description**

MDI frame and sheet windows

**Syntax**

```plaintext
window.SetToolbarPos( toolbarindex, x, y, width, height )
```

**Argument** | **Description**  
--- | ---  
`window` | The MDI frame or sheet to which the toolbar belongs  
`toolbarindex` | An integer whose value is the index of the toolbar whose settings you want to change  
`x` | An integer whose value is the x coordinate of the floating toolbar  
`y` | An integer whose value is the y coordinate of the floating toolbar  
`width` | An integer whose value is the width of the floating toolbar  
`height` | An integer whose value is the height of the floating toolbar

**Return value**

Integer. Returns 1 if it succeeds. `SetToolbarPos` returns -1 if there is no toolbar for the index you specify or if an error occurs. If any argument’s value is `null`, `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:

```plaintext
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:

```powershell
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:

```powershell
integer ix, iy, iw, ih

// Find out toolbar size
w_frame.GetToolbarPos(2, ix, iy, iw, ih)

// Set the position, taking the size into account
w_frame.SetToolbarPos(2, &
    w_frame.WorkspaceWidth() - iw, &
    w_frame.WorkspaceHeight() &
    - ih - w_frame.MDI_1.MicroHelpHeight, &
    iw, ih)

// Set the alignment to floating
w_frame.SetToolbar(2, TRUE, Floating!)
```

See also
GetToolbar
SetToolbar
SetToolbarPos
SetTop

Scrolls a list box control so that the specified item is the first visible item.

Applies to
ListBox and PictureListBox controls

Syntax

\[ \text{listboxname}.\text{SetTop}(\text{index}) \]

<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 \( \text{lb\_Actions} \) to the top of the ListBox so that it is the first visible item:

\[ \text{lb\_Actions}.\text{SetTop}(6) \]

The following statement scrolls the currently selected item in \( \text{lb\_Actions} \) to the top of the list of items:

\[ \text{lb\_Actions}.\text{SetTop}(\text{lb\_Actions.SelectedIndex()}) \]

See also
SetFocus
SetState

SetTraceFileName

Specifies the name of the trace file PowerBuilder will analyze when the BuildModel function is called.

Applies to
Profiling and TraceTree objects

Syntax

\[ \text{instancename}.\text{SetTraceFileName}(\text{tracefilename}) \]

<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
- **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
SetTransPool

**Description**
Sets up a pool of database transactions for an application. SetTransPool allows you to minimize the overhead associated with database connections and also limit the total number of database connections permitted.

**Applies to**
Application object

**Syntax**
```
applicationname.SetTransPool ( minimum, maximum, timeout )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationname</td>
<td>The name of the application object for which you want to establish a transaction pool</td>
</tr>
<tr>
<td>minimum</td>
<td>The minimum number of transactions to be kept open in the pool</td>
</tr>
<tr>
<td>maximum</td>
<td>The maximum number of transactions that can be open in the pool</td>
</tr>
<tr>
<td>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.

**Usage**
Transaction pooling maximizes database throughput while also controlling the number of database connections that can be open at one time. By establishing a transaction pool before connecting to the database, an application can reuse connections made to the same data source.

When an application connects to a database without using transaction pooling, PowerBuilder physically terminates each database transaction for which a DISCONNECT statement is issued. When transaction pooling is in effect, PowerBuilder logically terminates the database connections and commits any database changes, but does not physically remove them. Instead, the database connections are kept open in the transaction pool so that they can be reused for other database operations.

Before reusing a connection in the transaction pool, PowerBuilder checks to see that the database parameters specified in the incoming connection request match those specified by one of the connections in the pool. A match occurs when both transaction objects specify the same DBMS, ServerName, LogID, LogPass, Database, and DBParm values.

The minimum value specified in the SetTransPool function must be less than or equal to the maximum value. When the minimum value is less than the maximum and the number of transactions in the pool is greater than the minimum, PowerBuilder physically terminates connections for which a DISCONNECT statement is issued until the minimum number is reached.
The maximum value specified for a transaction pool limits the total number of database connections made by the application. When the transaction pool is full, each attempt to connect will fail after the timeout interval has been exceeded.

To set up transaction pooling for an application, you need to issue `SetTransPool` before establishing any connections to the database.

**Examples**

A distributed PowerBuilder server application that services a high volume of short database transactions to the same data source could issue the following statement in its application Open event:

```
server_app.SetTransPool(12, 16, 10)
```

This statement specifies that up to 16 database connections will be supported through this server, and that 12 connections will be kept open once successfully connected. When the maximum number of connections has been reached, each subsequent connection request will wait for up to 10 seconds for a connection in the pool to become available. After 10 seconds, the connection will fail but no error is returned. You can use the `DBHandle` function to determine whether a connection is valid.

The following statement specifies that up to 8 database connections will be allowed through this server, and that all 8 will be kept open once successfully connected. When the transaction pool is full, each subsequent connection request will immediately fail:

```
server_app.SetTransPool(8, 8, 0)
```

**See also**

`DBHandle`

`SetTrans` method for DataWindows in the *DataWindow Reference* or the online Help

`SetTransObject` method for DataWindows in the *DataWindow Reference* or the online Help
SetValue

Sets the date and time in the Value property of the control.

Applies to

DatePicker control

Syntax

controlname.SetValue (d, t)

colorname.SetValue (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 set the date and time</td>
</tr>
<tr>
<td>d</td>
<td>The date value to be set in the Value property</td>
</tr>
<tr>
<td>t</td>
<td>The time value to be set in the Value property</td>
</tr>
<tr>
<td>dt</td>
<td>The DateTime value to be set in the Value property</td>
</tr>
</tbody>
</table>

Return value

Integer. Returns 1 for success and one of the following negative values for failure:

-1 The value cannot be set
-2 Other error

Usage

The SetValue function can set the Value property using separate date and time variables or a single DateTime variable.

Examples

This example sets the Value property of a DatePicker control using separate date and time values:

date d
time t
d=date("2007/12/27")
t=time("12:00:00")

dp_1.SetValue(d, t)

This example sets the Value property using a DateTime value:

date d
time t
datetime dt
dt = DateTime(d, t)

dp_1.SetValue(dt)

See also

GetText
GetValue
**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>instancenames</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>classnames</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:

```powershell
integer status
string InstanceNames[]
string ClassNames[]

status = SharedObjectDirectory(InstanceNames, & ClassNames)
```

**See also**
SharedObjectGet
SharedObjectRegister
**SharedObjectGet**

**Description**

Gets a reference to a shared object instance.

**Syntax**

```plaintext
SharedObjectGet ( instancename, objectinstance )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instancename</td>
<td>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 <code>SharedObjectRegister</code> function.</td>
</tr>
<tr>
<td>objectinstance</td>
<td>An object variable of type <code>PowerObject</code> in which you want to store an instance of a shared object.</td>
</tr>
</tbody>
</table>

**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 object. 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:

```powerScript
// 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:

```powerScript
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:

```powerScript
// 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 on the callback object, passing it the DataWindow blob returned from the server component:

```powershell
blob lblb_data
long ll_rv
ll_rv = iuo_employee.retrievedata(lblb_data)
auo_callback.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)
```

**Argument** | **Description**
--- | ---
classname | The name of the user object that you want to share
instancename | A string whose value is the name you want to assign to the shared object instance

**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**
```
SharedObjectUnregister (instancename)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instancename</code></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`:
```
SharedObjectUnregister("share1")
```

**See also**
`SharedObjectRegister`

---

**Show**

**Description**
Makes an object or control visible, if it is hidden. If the object is already visible, `Show` brings it to the top.

**Applies to**
Any object

**Syntax**
```
objectname.Show()
```

<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 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.
CHAPTER 10  PowerScript Functions

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:

```powerscript
objectname.Visible = true
```

This statement:

```powerscript
m_status.m_options.Visible = TRUE
```

is equivalent to:

```powerscript
m_status.m_options.Show()
```

Examples

This statement makes visible the menu selection called `m_options` on the menu `m_status`:

```powerscript
m_status.m_options.Show()
```

This statement makes the child window `w_child` visible:

```powerscript
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**

```powerscript
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>
<tr>
<td><code>editheadfoot</code></td>
<td>A boolean value specifying the editing panel to display. Values are:</td>
</tr>
<tr>
<td></td>
<td>•  TRUE – Display the header and footer editing panels</td>
</tr>
<tr>
<td></td>
<td>•  FALSE – Display the detail editing panel for the document body</td>
</tr>
</tbody>
</table>

---
ShowHeadFoot

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:

```powerbuilder
rte_1.ShowHeadFoot(TRUE, 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 } )
```

<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>
</tbody>
</table>
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:

   ShowHelp("C:\PB\INQ.HLP", Index!)

This statement displays Help topic 143 in the file EMP.HLP file:

   ShowHelp("EMP.HLP", Topic!, 143)

This statement displays the Help topic associated with the keyword Part# in the file EMP.HLP:

   ShowHelp("EMP.HLP", Keyword!, "Part#")
**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.HELP", 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**
```
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):
```
Sign(5)
```
This statement returns 0 (zero has no sign):
```
Sign(0)
```
This statement returns -1 (the number is negative):
```
Sign(-5)
```

**See also**
`Sign` method for DataWindows in the *DataWindow Reference* or online Help

### SignalError

**Description**
Causes a SystemError event at the application level.

**Syntax**
```
SignalError ( { number }, { text } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</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</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.
SignalError

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 User’s Guide`.

Examples

These statements set values in the Error object and then trigger a SystemError event so the error processing for these values can be tested:

```powershell
int error_number
string error_text
Error.Number = 1010
Error.Text = "Salary must be a positive number."
Error.Windowmenu = "w_emp"

error_number = Error.Number
error_text = Error.Text

SignalError(error_number, error_text)
```

See also

`PopulateError`
**Sin**

**Description**
Calculates the sine of an angle.

**Syntax**
```
Sin ( n )
```

**Argument** | **Description**
--- | ---
`n` | The angle (in radians) for which you want the sine

**Return value**
Double. Returns the sine of `n`. If `n` is null, `Sin` returns null.

**Examples**
This statement returns .8414709848078965:
```
Sin(1)
```
This statement returns 0:
```
Sin(0)
```
This statement returns 0:
```
Sin(Pi(1))
```

**See also**
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**
```
Sleep ( seconds )
```

**Argument** | **Description**
--- | ---
`seconds` | Long for the number of seconds you want the application to pause

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

**Examples**
This example pauses the application for 5 seconds:
```
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.

Syntax

<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)
```

Argument	Description
---
`treeviewname` The name of the TreeView control in which you want to sort items.
`itemhandle` The item for which you want to sort its children.
`sorttype` The sort method you want to use. Valid values are:
- Ascending!
- Descending!
- UserDefinedSort!

Return value

Integer. Returns 1 if it succeeds and -1 if it fails.

Usage

The Sort function only sorts the immediate level beneath the specified item. If you want to sort multiple levels, use SortAll. If you specify UserDefinedSort! as your `sorttype`, define your sort criteria in the Sort event of the TreeView control. 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
CHAPTER 10  PowerScript Functions

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:</td>
</tr>
<tr>
<td></td>
<td>Ascending!</td>
</tr>
<tr>
<td></td>
<td>Descending!</td>
</tr>
<tr>
<td></td>
<td>Unsorted!</td>
</tr>
<tr>
<td></td>
<td>UserDefinedSort!</td>
</tr>
<tr>
<td>column</td>
<td>The number of the column by which you wish to sort the ListView items. (optional)</td>
</tr>
</tbody>
</table>

Return value  Integer. Returns 1 if it succeeds and -1 if it fails.

Usage  The default sort is alphanumeric.

Examples  If you do not specify a column to sort, the first column is sorted.

This example sorts the items in column three of a ListView:

```powerscript
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>
</tbody>
</table>

PowerScript Reference Volume 2  1043
SortAll

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
</table>
| sorttype | The sort method you want to use. Values are:  
|          | Ascending!  
|          | Descending! |
|          | Unsorted!   
|          | UserDefinedSort! |

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

Usage If you specify UserDefinedSort! as your sorttype, define your sort criteria in the Sort event of the TreeView control.

The SortAll function 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:

```powershell
long ll_tvi

//Find the current treeitem
ll_tvi = tv_list.GetItem(CurrentTreeItem! , 0)

//Sort all children
tv_list.SortAll(ll_tvi , Ascending!)
```

This example recursively sorts the entire TreeView control:

```powershell
long ll_tvi

//Find the root treeitem
ll_tvi = tv_list.GetItem(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>n</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:
```
string Name
Name = Space(4)
```
This statement assigns 40 spaces to the string Name:
```
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
```
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.

Examples
This statement returns 1.414213562373095:
```
Sqrt(2)
```
This statement results in an error at execution time:
```
Sqrt(-2)
```
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.

Applies to Pipeline objects

Syntax

`pipelineobject.Start ( sourcetrans, destinationtrans, errorobject {, arg1, arg2,..., argn })`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pipelineobject</code></td>
<td>The name of a pipeline user object that contains the pipeline object to be executed</td>
</tr>
<tr>
<td><code>sourcetrans</code></td>
<td>The name of a transaction object with which to connect to the source database</td>
</tr>
<tr>
<td><code>destinationtrans</code></td>
<td>The name of a transaction object with which to connect to the target database</td>
</tr>
<tr>
<td><code>errorobject</code></td>
<td>The name of a DataWindow control or Data Store in which to store the pipeline error DataWindow</td>
</tr>
<tr>
<td><code>argn</code> (optional)</td>
<td>One or more retrieval arguments as specified for the pipeline object in the Data Pipeline painter</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

See also `Sqrt` method for `DataWindows` in the `DataWindow Reference` or online Help
-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 User’s Guide. For more information on using a pipeline in an application, see Application Techniques.

When you dynamically assign the pipeline object to the user object’s DataObject property, you must remember to include the pipeline object in a dynamic library when you build your application’s executable.

Examples

The following script creates an instance of the pipeline user object, assigns a pipeline object to the pipeline user object’s DataObject property, and executes the pipeline. \( I_{src}\) and \( I_{dst}\) are transaction objects that have been previously declared and created. Another script has established the database connections.

\( U_{pipe} \) is the user object inherited from the pipeline system object. \( I_{upipe} \) is an instance variable of type \( u_{pipe} \). \( P_{pipe} \) is a pipeline object created in the Data Pipeline painter:

\[
\begin{align*}
I_{upipe} &= \text{CREATE} \quad u_{pipe} \\
I_{upipe}.\text{DataObject} &= "p_{pipe}"
\end{align*}
\]

\[
I_{upipe}.\text{Start}(I_{src}, I_{dst}, dw_1)
\]

See also

Cancel
Repair

### Syntax 2

**For activating timing objects**

Activates a timing object causing a Timer event to occur repeatedly at the specified interval.

**Applies to**

Timing objects

**Syntax**

\[
timingobject.\text{Start}(\ interval \ )
\]

<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 ( \text{interval} ) 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:

```
integer li_err
li_err = This.of_connect()
IF li_err <> 1 THEN
  MessageBox("Timer Error", "Connection failed ")
END IF
```

When the main window (`w_timer`) opens, its Open event script registers the `uo_timing` user object as a shared object:

```
ErrorReturn result
string ls_result
```
SharedObjectRegister("uo_timing","Timing")
result = SharedObjectGet("Timing", iuo_timing)
// convert enumerated type to string
ls_result = of_converteror(result)

IF result = Success! THEN
    sle_stat.text = "Object Registered"
ELSE
    MessageBox("Failed", "SharedObjectGet failed, " & + "Status code: "+ls_result)
END IF

The Start Timer button starts the timer with an interval of five seconds:

double ld_interval
integer li_err

IF (isvalid(iuo_timing)) THEN
    li_err = iuo_timing.Start(5)
    ld_interval = iuo_timing.interval
    sle_2.text = "Timer started. Interval is " & + string(ld_interval) + " seconds"
    // disable Start Timer button
    THIS.enabled = FALSE
ELSE
    sle_2.text = "No timing object"
END IF

The Get Hits button calls the of_hitcount function and writes the result in a single line edit:

long ll_hits

IF (isvalid(iuo_timing)) THEN
    ll_hits = iuo_timing.of_hitcount()
    sle_hits.text = string(ll_hits)
ELSE
    sle_hits.text = ""
    sle_stat.text = "Invalid timing object..."
END IF

The Stop Timer button stops the timer, reenables the Start Timer button, and resets the hit counter:

integer li_err

IF (isvalid(iuo_timing)) THEN
    li_err = iuo_timing.Stop()
IF li_err = 1 THEN
  sle_stat.text = "Timer stopped"
  cb_start.enabled = TRUE
  iuo_timing.of_resetcounter()
ELSE
  sle_stat.text = "Error - timer could not be stopped"
END IF

ELSE
  sle_stat.text = "Error - no timing object"
END IF

The Close button checks that the timer has been stopped and closes the window if it has:

IF iuo_timing.running = TRUE THEN
  MessageBox("Error","Click the Stop Timer button to clean up before closing")
ELSE
  close(parent)
END IF

The Close event for the window unregisters the shared timing object:

SharedObjectUnregister("Timing")

The of_converter window function converts the ErrorReturn enumerated type to a string. It takes an argument of type ErrorReturn:

string ls_result

CHOOSE CASE a_error
CASE Success!
  ls_result = "The function succeeded"
CASE FeatureNotSupportedError!
  ls_result = "Not supported on this platform"
CASE SharedObjectExistsError!
  ls_result = "Instance name already used"
CASE MutexCreateError!
  ls_result = "Locking mechanism unobtainable"
CASE SharedObjectCreateInstanceError!
  ls_result = "Object could not be created"
CASE SharedObjectCreatePBSessionError!
  ls_result = "Could not create context session"
CASE SharedObjectNotExistsError!
  ls_result = "Instance name not registered"
CASE ELSE
\begin{verbatim}
ls_result = "Unknown Error Code"
END CHOOSE

RETURN ls_result
\end{verbatim}

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 )
```

<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>
<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>
</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*:

```
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:

```
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**

<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.</td>
</tr>
<tr>
<td>applname</td>
<td>The DDE name for your application.</td>
</tr>
<tr>
<td>topic</td>
<td>A string whose value is the basic data grouping the DDE client application references.</td>
</tr>
<tr>
<td>item (optional)</td>
<td>A comma-separated list of one or more strings (data within topic) that specify what your DDE server application supports (for example, &quot;Table1&quot;,&quot;Table2&quot;).</td>
</tr>
</tbody>
</table>

**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</td>
<td>Obtain the command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetCommandDDEOrigin</td>
<td>Find out what client application sent the command</td>
</tr>
<tr>
<td>Sends data</td>
<td>RemoteSend</td>
<td>GetDataDDE</td>
<td>Obtain the data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetDataDDEOrigin</td>
<td>Find out what client application sent the data</td>
</tr>
<tr>
<td>Requests data from your server</td>
<td>RemoteRequest</td>
<td>SetDataDDE</td>
<td>Send the requested data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RespondRemote</td>
<td>Acknowledge the request</td>
</tr>
<tr>
<td>Wants to terminate the hot link</td>
<td>RemoteHotLinkStop</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Examples**

This statement causes your PowerBuilder application to begin acting as a server. It is known to other DDE applications as *MyPBApp*; its topic is *System*, and it supports items called *Table1* and *Table2*:

```
StartServerDDE(w_emp, "MyPBApp", "System", &
               "Table1", "Table2")
```

**See also**

StopServerDDE

---

**State**

**Description**

Determines whether an item in a ListBox control is highlighted.

**Applies to**

ListBox and PictureListBox controls

**Syntax**

```
listboxname.State( index )
```

**Argument** | **Description**
--- | ---
listboxname | The name of the ListBox or PictureListBox in which you want to obtain the state (highlighted or not highlighted) of the item identified by index
index | The number of the item for which you want to obtain the state
State

Return value

Integer. Returns 1 if the item in listboxname identified by index is highlighted and 0 if it is not. If the index does not point to a valid item number, State returns -1. If any argument’s value is null, State returns null.

Usage

The State and SetState functions are meant for a ListBox that allows multiple selections (its MultiSelect property is true). To find all of a list’s selected items, loop through the list, checking the state of each item.

The SelectedItem and SelectItem functions are meant for single-selection 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:

integer li_Item
li_Item = lb_Contact.State(3)

The following statements obtain the text of all the selected items in a ListBox that allows the user to select more than one item. The MessageBox function displays each item as it is found. You could include other processing that created an array or list of the selected values:

integer li_ItemTotal, li_ItemCount

// Get the number of items in the ListBox.
li_ItemTotal = lb_contact.TotalItems( )

// Loop through all the items.
FOR li_ItemCount = 1 to li_ItemTotal
  // Is the item selected? If so, display the text
  IF lb_Contact.State(li_ItemCount) = 1 THEN &
    MessageBox("Selected Item", &
    lb_Contact.text(li_ItemCount))
  NEXT

This statement executes some statements if item 3 in the ListBox lb_Contact is highlighted:

IF lb_Contact.State(3) = 1 THEN ...

See also

SelectedItem
SetState
**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
```
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>
**Stop**

**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><code>timingobject</code></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

**Syntax 2**  
**For stopping an animation from playing**

**Description**  
Stops an animation (an AVI clip) from playing.

**Applies to**  
Animation controls

**Syntax**

```
animationname.Stop ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>animationname</code></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**

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><code>location</code></td>
<td>A string whose value is the location at which you want to end the hot link, as specified in the <strong>StartHotLink</strong> function that established the link</td>
</tr>
<tr>
<td><code>applname</code></td>
<td>A string whose value is the DDE name of the server application, as specified in the <strong>StartHotLink</strong> function</td>
</tr>
<tr>
<td><code>topic</code></td>
<td>A string identifying the data or the instance of the application in which the hot link is stopped, as specified in the <strong>StartHotLink</strong> function</td>
</tr>
</tbody>
</table>

**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

```powerbuilder
StopServerDDE ( { windowname, } applname, topic )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windowname</code></td>
<td>(optional) The name of the server window. The default is the current window. If you have more than one server window, <code>windowname</code> is required.</td>
</tr>
<tr>
<td><code>applname</code></td>
<td>The DDE name for your PowerBuilder application.</td>
</tr>
<tr>
<td><code>topic</code></td>
<td>A string whose value is the topic you declared when you called <code>StartServerDDE</code></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:

```powerbuilder
StopServerDDE (w_emp, "MyPBApp", "System")
```

See also
`StartServerDDE`

String

String has two syntaxes.

<table>
<thead>
<tr>
<th>To</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format data as a string according to a specified display format mask</td>
<td>Syntax 1</td>
</tr>
<tr>
<td>Convert a blob to a string</td>
<td>Syntax 2</td>
</tr>
</tbody>
</table>
### Syntax 1

#### Description
Formats data, such as time or date values, according to a format mask. You can convert and format date, DateTime, numeric, and time data. You can also apply a display format to a string.

#### Syntax

```plaintext
String { data, { format } }
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</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>format</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>

| 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:

```plaintext
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 User’s Guide*. Note that, although a format can include color specifications, the colors are ignored when you use String in PowerScript. Colors appear only for display formats specified in the DataWindow painter.

If the display format 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:

```powershell
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:

```
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:

```
Date order_date = 2002-06-11
string date1
date1 = String(order_date,"m-d-yy")
```

This example includes a format for a null date value so that when `order_date` is null, `date1` is set to `none`:

```
Date order_date = 2002-06-11
string date1
SetNull(order_date)
date1 = String(order_date, "m-d-yy;'none'")
```

This statement applies a format to a DateTime value and returns Jan 31, 2001 6 hrs and 8 min:

```
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:

```
DateTime sys_datetime
string datetime1
sys_datetime = DateTime(Today(), Now())
sle_date.text = String(sys_datetime, &
"m-d-yy h:mm am/pm;'none'")
```

This statement applies a format to a numeric value and returns $5.00:

```
String(5,"$#,##0.00")
```

These statements set `string1` to 0123:

```
integer nbr = 123
string string1
string1 = String(nbr,"0000;(000);****;empty")
```

These statements set `string1` to (123):

```
integer nbr = -123
string string1
string1 = String(nbr,"000;(000);****;empty")
```
These statements set \textit{string1} to ****:

\begin{verbatim}
integer nbr = 0
string string1
string1 = String(nbr,"0000;(000);****;empty")
\end{verbatim}

These statements set \textit{string1} to "empty":

\begin{verbatim}
integer nbr
string string1
SetNull(nbr)
string1 = String(nbr,"0000;(000);****;empty")
\end{verbatim}

This statement formats string data and returns \textit{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:

\begin{verbatim}
String("ABC", "@-@-@")
\end{verbatim}

This statement returns A*B:

\begin{verbatim}
String("ABC", "@*@")
\end{verbatim}

This statement returns ABC:

\begin{verbatim}
String("ABC", "@@@")
\end{verbatim}

This statement returns a space:

\begin{verbatim}
String("ABC", " ")
\end{verbatim}

This statement applies a display format to time data and returns 6 hrs and 8 min:

\begin{verbatim}
String(06:08:02,'h "hrs and" m "min"')
\end{verbatim}

This statement returns 08:06:04 pm:

\begin{verbatim}
String(20:06:04,"hh:mm:ss am/pm")
\end{verbatim}

This statement returns 8:06:04 am:

\begin{verbatim}
String(08:06:04,"h:mm:ss am/pm")
\end{verbatim}

See also String method for DataWindows in the \textit{DataWindow Reference} or online Help.
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.

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:

```powerscript
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:

```powerscript
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

### 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>jaguarorb</td>
<td>An instance of JaguarORB.</td>
</tr>
<tr>
<td>objstring</td>
<td>A string that represents a CORBA object. 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.</td>
</tr>
<tr>
<td>object</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` 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://machinea:9000
```

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")
my_orb.String_To_Object("iiop://myhost:9000", &
my_corbaobj)

my_corbaobj._narrow(my_manager, &
"SessionManager/Manager")
my_session = my_manager.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)
```
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:

```power_script
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_orb.String_to_Object("iiop://svr1:9000", &
  my_corbaObj)
my_corbaObj._narrow(my_mgr, "SessionManager/Manager")
my_Session = my_mgr.createSession("jagadmin", ")
my_corbaObj = my_session.lookup("Cart")
ll_return = my_corbaObj._narrow(my_CartHome,
  "com/shopping/CartHome")

my_corbaObj = my_CartHome.create()
my_Cart.addItem()
```

See also
Init
Lookup
_Narrow
Object_To_String
Resolve_Initial_References
**SuspendTransaction**

**Description**
Suspend the EAServer transaction associated with the calling thread.

**Applies to**
CORBACurrent objects

**Syntax**

```
CORBACurrent.SuspendTransaction()
```

**Argument**

<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**
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. 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:

```// 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

1070 PowerBuilder
CHAPTER 10  PowerScript Functions

**SyntaxFromSQL**

Generates DataWindow source code based on a SQL SELECT statement.

**Applies to**  
Transaction objects

**Syntax**

```
transaction.SyntaxFromSQL (sqlselect, presentation, err)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>transaction</strong></td>
<td>The name of a connected transaction object.</td>
</tr>
<tr>
<td><strong>sqlselect</strong></td>
<td>A string whose value is a valid SQL SELECT statement.</td>
</tr>
<tr>
<td><strong>presentation</strong></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><strong>err</strong></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:

```java
sqlca.autocommit=True
ls_dws=sqlca.syntaxfromsql (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:

```java
"Style ( Type=value property=value ... )
DataWindow ( property=value ... )
Column ( property=value ... )
Group groupby_column1 Fby_column2 ... 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:

```powerbuild
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:

```powerbuild
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
`instancename.SystemRoutine ( theroutine )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instancename</code></td>
<td>Instance name of the Profiling object.</td>
</tr>
<tr>
<td><code>theroutine</code></td>
<td>A value of type ProfileRoutine containing the routine node representing the system root. 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
Use this function to extract the routine node representing the system root in a performance analysis model. You must have previously created the performance analysis model from a trace file using the `BuildModel` function. The routine node is defined as a ProfileRoutine object and provides the time spent in the routine, any called routines, the number of times each routine was called, and the class to which the routine belongs.

Examples
This example provides the routine that represents the system root in a performance analysis model:

```
Profiling lpro_model
ProfileRoutine lprort_routine

lpro_model.BuildModel()
lpro_model.SystemRoutine(lprort_routine)
...
```

See also
`BuildModel`
CHAPTER 10  PowerScript Functions

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

\texttt{tabcontrolname.TabPostEvent \{ event \{ word, long \} \}}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{tabcontrolname}</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>\texttt{event}</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 \texttt{tabcontrolname} and a script must exist for the event in \texttt{tabcontrolname}.</td>
</tr>
<tr>
<td>\texttt{word} (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 \texttt{long}, but not \texttt{word}, enter 0. (For cross-platform compatibility, WordParm and LongParm are both longs).</td>
</tr>
<tr>
<td>\texttt{long} (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 \texttt{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 \texttt{tab_address} contains several tab pages inherited from \texttt{uo_list} and \texttt{uo_list} has a user event called \texttt{ue_display}. This statement posts the event \texttt{ue_display} for each the tab pages in \texttt{tab_address}:

\texttt{tab_address.TabPostEvent("ue_display")}

See also
TabTriggerEvent

PowerScript Reference Volume 2 1075
**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**
```
tagcontrolname.TabTriggerEvent ( event {, word, long} )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tagcontrolname</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>event</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 tagcontrolname and a script must exist for the event in tagcontrolname.</td>
</tr>
<tr>
<td>word (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 long, but not word, enter 0. (For cross-platform compatibility, WordParm and LongParm are both longs).</td>
</tr>
<tr>
<td>long (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 listboxname identified by index. If the index does not point to a valid item number, Text returns the empty string (“”). If any argument’s value is null, Text returns null.
Examples

Assume the ListBox lb_Cities contains:

- Atlanta
- Boston
- Chicago
- Denver

Then these statements store the text of item 3, which is Chicago, in current_city:

```cpp
string current_city
current_city = lb_Cities.Text(3)
```

See also

FindItem
SelectedItem
SelectedText

---

**TextLine**

**Description**

Obtains the text of the line that contains the insertion point. TextLine works for controls that can contain multiple lines.

**Applies to**

DataWindow, EditMask, MultiLineEdit, and RichTextEdit controls

**Syntax**

`editname.TextLine()`

**Argument** | **Description**  
--- | ---  
`editname` | The name of the DataWindow control, EditMask, MultiLineEdit, or RichTextEdit 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:

```cpp
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

Extracts a time value from a DateTime value or a blob.

**Syntax**

```
Time ( datetime )
```

**Argument**

- `datetime`: A DateTime value or a blob in which the first value is a time or DateTime value. The rest of the contents of the blob is ignored. `DateTime` can also be an Any variable containing a DateTime or blob.

**Return value**

Time. Returns the time in `datetime` as a time. If `datetime` does not contain a valid time or is an incompatible 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`:

```powerscript
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**

**Description**

Converts a string containing a valid time into a time value.

**Syntax**

```
    Time ( string )
```

**Argument** | **Description**
--- | ---
`string` | A string whose value is a valid time (such as 8am or 10:25) that you want returned as a time. Only the hour is required; you do not have to include the minutes, seconds, or microseconds of the time or am or pm. The default value is 00 for minutes and seconds and 000000 for microseconds. PowerBuilder determines whether the time is am or pm based on a 24-hour clock. String can also be an Any variable containing a string or blob.

**Return value**

Time. Returns the time in string as a time. If string does not contain a valid time or is an incompatible 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:

```powershell
    Time("23:59:15")
```
This statement converts the text in the SingleLineEdit `sle_Time_Received` to a time value:

```
Time(sle_Time_Received.Text)
```

**Syntax 3**

**For integers**

Description

Combines integers representing hours, minutes, seconds, and microseconds into a time value.

Syntax

```
Time ( hour, minute, second {, microsecond } )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hour</td>
<td>The integer for the hour (00 to 23) of the time</td>
</tr>
<tr>
<td>minute</td>
<td>The integer for the minutes (00 to 59) of the time</td>
</tr>
<tr>
<td>second</td>
<td>The integer for the seconds (0 to 59) of the time</td>
</tr>
<tr>
<td>microsecond</td>
<td>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) (optional)</td>
</tr>
</tbody>
</table>

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.

Examples

These statements set `What_Time` to a time value with microseconds, and display the resulting time as a string in `st_1`. The default display format does not include microseconds, so the `String` function specifies a display format with microseconds. Leading zeros are appended to the string value for microseconds:

```
Time What_Time
What_Time = Time(10, 15, 45, 234)
st_1.Text = String(What_Time, "hh:mm:ss:fffffff")
```

The time in the string variable is set to 10:15:45:000234.

These statements set `What_Time` to 10:15:45:

```
Time What_Time
What_Time = Time(10, 15, 45)
```

See also

Time method for DataWindows in the *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**

```
Timer ( interval[, windowname] )
```

**Argument** | **Description**
--- | ---
`interval` | 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 `interval` is 0, `Timer` turns off the timer so that it no longer triggers Timer events.

`windowname` (optional) | The window in which you want the timer event to be triggered. The window must be an open window. If you do not specify a window, the Timer event occurs in the current window.

**Return value**
Integer. Returns 1 if succeeds and -1 if an error occurs. If any argument’s value is null, `Timer` returns null.

**Usage**
Do not call the `Timer` function in the Timer event. The timer gets reset automatically and the Timer event retrigger at the interval that has already been established. Call the `Timer` function in another event’s script when you want to stop the timer or change the interval.

**Examples**
This statement triggers a Timer event every two seconds in the active window:

```
Timer (2)
```

This statement stops the triggering of the Timer event in the active window:

```
Timer (0)
```

These statements trigger a Timer event every half second in the window `w_Train`:

```
Open (w_Train)
Timer (0.5, w_Train)
```

This example causes the current time to be displayed in a StaticText control in a window. Calling `Timer` in the window’s Open event script starts the timer. The script for the Timer event refreshes the displayed time.

In the window’s Open event script, the following code displays the time initially and starts the timer:

```
st_time.Text = String (Now (), "hh:mm")
Timer (60)
```
In the window’s Timer event, which is triggered every minute, this code displays the current time in the StaticText `st_time`:
```
st_time.Text = String(Now(), "hh:mm")
```

See also

Idle

---

**Toansi**

**Description**
Converts a character string to an ANSI blob.

**Syntax**

```
Toansi ( 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 an ANSI blob</td>
</tr>
</tbody>
</table>

**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 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

---

PowerScript Reference Volume 2

1083
**Today**

**Description**
Obtains the system date and, in some cases, the system time.

**Syntax**
`Today()`

**Return value**
Date. Returns the current system date.

**Usage**
Although the 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:

```powerbuilder
Today()
```

This statement executes some statements when the current system date is before April 15, 2003:

```powerbuilder
IF Today() < 2003-04-15 THEN ...
```

This statement displays the current date in the `StaticText` `st_date` in the corner of a window:

```powerbuilder
st_date.Text = String(Today(), "m/d/yy")
```

This statement displays the current date and time in the `StaticText` `st_date`:

```powerbuilder
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()`

**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:

```powerscript
integer Num
Num = lb_Contacts.Top()
```

If the user has not scrolled the list in `lb_Contacts`, then `Num` is set to 1:

```powerscript
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:

```powerscript
integer Num
Num = lb_Contacts.SelectedIndex()
IF lb_Contacts.Top() <> Num THEN &
    lb_contacts.SetTop(Num)
```

**See also**

SelectedIndex
SetTop

<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 index of the first visible item in the list</td>
</tr>
</tbody>
</table>
**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><code>listviewname</code></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:

```powerbuilder
int li_cols
li_cols = lv_list.TotalColumns()
sle_info.text = "Total columns = " + string(li_cols)
```

**See also**
TotalItems
TotalSelected

---

**TotalItems**

**Description**
Determines the total number of items in a ListBox control.

**Applies to**
ListBox, DropDownListBox, PictureListBox, DropDownPictureListBox, and ListView controls

**Syntax**
`listcontrolname.TotalItems ( )`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listcontrolname</code></td>
<td>The name of the ListBox, DropDownListBox, PictureListBox, DropDownPictureListBox, or ListView in which you want the total number of items</td>
</tr>
</tbody>
</table>

**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:

```powerbuilder
integer Total
Total = lbx_Actions.TotalItems()
```
This FOR loop is executed for each item in lb_Actions:

```powerscript
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()
```

**Argument**
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listcontrolname</td>
<td>The name of the ListBox, PictureListBox, or ListView in which you want the number of items that are selected</td>
</tr>
</tbody>
</table>

**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:

```powerscript
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:

```powerscript
IF lb_Actions.TotalSelected() > 3 THEN
    MessageBox("Warning", &
       "You can only select 3 items!")
ELSE
    ... // Some processing
END IF
```

See also TotalItems
**ToUnicode**

**Description**
Converts 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.

**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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>A read-only string, logged to the trace file, used to identify a tracing block. If <code>identifier</code> is null, an empty string is placed in the trace file.</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!** – `TraceBegin` has already been called
CHAPTER 10  PowerScript Functions

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:

```
TimerKind ltk_kind

CHOOSE CASE ddlb_timestamp.text
    CASE "None"
        ltk_kind = TimerNone!
    CASE "Clock"
        ltk_kind = Clock!
    CASE "Process"
        ltk_kind = Process!
    CASE "Thread"
        ltk_kind = Thread!
END CHOOSE

TraceOpen(sle_filename.text,ltk_kind)
TraceEnableActivity(ActESQL!)
TraceEnableActivity(ActGarbageCollect!)
TraceEnableActivity(ActObjectCreate!)
TraceEnableActivity(ActObjectDestroy!)

**TraceBegin**("Trace_block_1")
```

See also

**TraceOpen**
**TraceEnableActivity**
**TraceEnd**
**TraceClose**

Description
Closes the trace file.

Syntax
```
TraceClose()
```

Return value
ErrorReturn. Returns one of the following values:
- Success! – The function succeeded
- FileNotOpenError! – TraceOpen has not been called yet
- FileCloseError! – The log file is full

Usage
TraceClose closes the trace file. If you have not already called TraceEnd, TraceClose will call that function before proceeding with its processing.

You typically include the TraceClose function in your application’s Close script.

Examples
This example stops logging of application trace activities and then closes the open trace file:
```
TraceEnd()
TraceClose()
```

See also
TraceBegin
TraceEnd
TraceOpen
TraceDisableActivity

Description
Disables logging of the specified trace activity.

Syntax
TraceDisableActivity ( activity )

<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

Return value
ErrorReturn. Returns one of the following values:

• Success! – The function succeeded

• FileNotOpenError! – TraceOpen has not been called yet

• TraceStartedError! – You have called TraceDisableActivity after TraceBegin and before TraceEnd

Usage
Use this function to disable the logging of the specified trace activities. You typically use this function if you are tracing only portions of an application run (and thus you are calling TraceBegin multiple times) and you want to log different activities during each portion of the application.

Unless specifically disabled with TraceDisableActivity, activities that were previously enabled with a call to the TraceEnableActivity function remain enabled throughout the entire application run.

You must always call the TraceEnd function before calling TraceDisableActivity.
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:

```plaintext
TraceEnableActivity(ActESQL!)
TraceEnableActivity(ActGarbageCollect)
TraceEnableActivity(ActObjectCreate!)
TraceEnableActivity(ActObjectDestroy!)

TraceBegin("Trace_block_1")

TraceEnd()

TraceDisableActivity(ActESQL!)
TraceDisableActivity(ActGarbageCollect!)

TraceBegin("Trace_block_2")
```

See also

TraceEnd
TraceEnableActivity
TraceEnableActivity

Description
Enables logging of the specified trace activity.

Syntax
TraceEnableActivity(activity)

Argument | Description
---|---
activity | A value of the enumerated datatype TraceActivity that identifies the activity to be logged. Values are:
  - ActError! – Occurrences of system errors and warnings
  - ActESQL! – Embedded SQL statement entry and exit
  - ActGarbageCollect! – Start and finish of garbage collection
  - ActLine! – Routine line hits (if this value is enabled, ActRoutine! is automatically enabled)
  - ActObjectCreate! – Object creation entry and exit
  - ActObjectDestroy! – Object destruction entry and exit
  - ActProfile! – Abbreviation for the ActRoutine!, ActESQL!, ActObjectCreate!, ActObjectDestroy, and ActGarbageCollect! values
  - ActRoutine! – Routine entry and exit
  - ActTrace! – Abbreviation for all activities except ActLine!

Return value
ErrorReturn. Returns one of the following values:
  - Success! – The function succeeded
  - FileNotOpenError! – TraceOpen has not been called yet
  - TraceStartedError! – You have called TraceEnableActivity after TraceBegin and before TraceEnd

Usage
Call the TraceEnableActivity function following the TraceOpen function. TraceEnableActivity allows you to specify the types of activities you want logged in the trace file. The default activity type logged is a user-defined activity type identified by the value ActUser!. This activity is enabled by the TraceOpen call. You must call TraceEnableActivity to specify the activities to be logged before you call TraceBegin.

Each call to TraceOpen resets the activity types to be logged to the default (that is, only ActUser! activities are logged).

Since the ActError! and ActUser! values require the passing of strings to the trace file, you must call the TraceError and TraceUser functions to log this information.
**TraceEnableActivity**

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:

```powershell
TimerKind lt_k_kind

CHOICE CASE ddlb_timestamp.text
  CASE "None"
    lt_k_kind = TimerNone!
  CASE "Clock"
    lt_k_kind = Clock!
  CASE "Process"
    lt_k_kind = Process!
  CASE "Thread"
    lt_k_kind = Thread!
END CHOICE

TraceOpen(sle_filename.text, lt_k_kind)

TraceEnableActivity(ActRoutine!)
TraceEnableActivity(ActESQL!)
TraceEnableActivity(ActGarbageCollect!)
TraceEnableActivity(ActError!)
TraceEnableActivity(ActCreateObject!)
TraceEnableActivity(ActDestroyObject!)

TraceBegin("Trace_block_1")
```

**See also**

- `TraceOpen`
- `TraceBegin`
- `TraceDisableActivity`
**TraceEnd**

**Description**
Inserts an activity type value in the trace file indicating that logging has ended and then stops logging application trace activities.

**Syntax**

```
TraceEnd()
```

**Return value**

ErrorReturn. Returns one of the following values:

- **Success!** – The function succeeded
- **FileNotFoundException!** – 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:

```
TraceEnd()
TraceClose()
```

**See also**

- TraceOpen
- TraceBegin
- TraceClose
- TraceDisableActivity
TraceError

Description
Logs your own error message and its severity level to the trace file if tracing of this activity type has been enabled.

Syntax
**TraceError ( severity, message )**

<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>

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:

```plaintext
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 wall 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>• TimeNone! – 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:

```
TimerKindltk_kind

CHOOSE CASE ddb_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!.

If `info` or `message` is null, `TraceUser` returns null and no entry is made in the log file.

**Usage**

`TraceUser` logs an activity type value of ActUser! to the trace file. This is the default activity type and is enabled when the `TraceOpen` function is called. You use the `TraceUser` function to record your own message identifying a specific occurrence during an application run. For example, you may want to log the occurrences of a specific return value or the beginning and end of a body of code. `TraceUser` works just like the `TraceError` function except that you use `TraceError` to identify more severe problems. The `info` and `message` values are passed without modification to the trace file.

**Examples**

This example logs user messages to the trace file identifying when a database retrieval is started and when it is completed:

```
dw_1.SetTransObject(SQLCA)

TraceUser(100, "Starting database retrieval")
IF dw_1.Retrieve() = -1 THEN
    TraceError(999, "Retrieve for dw_1 failed")
ELSE
    TraceUser(200, "Database retrieval complete")
END IF
```

**See also**

`TraceEnableActivity`  
`TraceError`
TriggerEvent

**Description**
Triggers an event associated with the specified object, which executes the script for that event immediately.

**Applies to**
Any object

**Syntax**
```
objectname.TriggerEvent( event, word, long )
```

<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>
<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>

**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.

1100
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:

```powerscript
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`.

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:

```powerscript
Send(Handle(Parent), 273, 0, Long(Handle(cb_OK), 0))
```

is equivalent to:

```powerscript
cb_OK.TriggerEvent(Clicked!)
```

**Examples**

This statement executes the script for the Clicked event in the CommandButton `cb_OK` immediately:

```powerscript
cb_OK.TriggerEvent(Clicked!)
```

This statement executes the script for the user-defined event `cb_exit_request` in the parent window:

```powerscript
Parent.TriggerEvent("cb_exit_request")
```

This statement executes the script for the Clicked event in the menu selection `m_File` on the menu `m_App`:

```powerscript
m_App.m_File.TriggerEvent(Clicked!)
```

**See also**

- `Post`
- `PostEvent`
- `Send`
TriggerPBEvent

Description
Triggers the specified user event in the child window contained in a PowerBuilder window ActiveX control.

Applies to
Window ActiveX controls

Syntax
activexcontrol.TriggerPBEvent ( name {, numarguments {, arguments } } )

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.");
}
PBRX1.ResetArgElements();
}

This VBScript example calls the TriggerPBEvent function:

Sub TrigEvent_OnClick()
    Dim retcd
    Dim myForm
    Dim args(1)
    Dim rc
    Dim numargs
    Dim theEvent
    retcd = 0
    numargs = 1
    rc = 0
    theEvent = "ue_args"
    Set myForm = Document.buttonForm
    args(0) = buttonForm.textToPB.value
    retcd = PBRX1.TriggerPBEvent(theEvent, & numargs, args)
    rc = PBRX1.GetLastReturn()
    if rc <> 1 then
        msgbox "Error. Empty string."
    end if
end sub

See also
GetLastReturn
SetArgElement
InvokePBFunction
**Trim**

**Description**
Removes leading and trailing spaces from a string.

**Syntax**
```
Trim(string)
```

**Argument** | Description
---|---
`string` | The string you want returned with leading and trailing spaces deleted.

**Return value**
String. Returns a copy of `string` with all leading and trailing spaces deleted if it succeeds and the empty string ("") if an error occurs. If `string` is null, `Trim` returns null.

**Usage**
`Trim` is useful for removing spaces that a user may have typed before or after newly entered data.

**Examples**
This statement returns BABE RUTH:
```
Trim(" BABE RUTH ")
```

This example removes the leading and trailing spaces from the user-entered value in the `SingleLineEdit` `sle_emp_fname` and saves the value in `emp_fname`:
```
string emp_fname
emp_fname = Trim(sle_emp_fname.Text)
```

**See also**
LeftTrim
RightTrim
`Trim` 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.

**Syntax**

```
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:
```
Truncate (x + 0.0000001, n)
```

**Examples**

This statement returns 9.2:
```
Truncate (9.22, 1)
```

This statement returns 9.2:
```
Truncate (9.28, 1)
```

This statement returns 9:
```
Truncate (9.9, 0)
```

This statement returns –9.2:
```
Truncate (–9.29, 1)
```

**See also**
Ceiling
Int
Round
Truncate method for DataWindows in the DataWindow Reference or online Help
Trust Verify

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 )

<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</td>
</tr>
<tr>
<td>reason</td>
<td>A long value indicating the reason for the call back. Values are:</td>
</tr>
<tr>
<td></td>
<td>• 1 REASON_CHAIN_INCOMPLETE</td>
</tr>
<tr>
<td></td>
<td>• 2 REASON_UNKNOWN_CA</td>
</tr>
<tr>
<td></td>
<td>• 3 REASON_CHAIN_EXPIRED</td>
</tr>
<tr>
<td></td>
<td>• 4 REASON_TRUSTDBPINNOTSET</td>
</tr>
<tr>
<td></td>
<td>• 5 REASON_TRUSTDBLOGINFAILED</td>
</tr>
</tbody>
</table>

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.

Examples

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:

```power
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  = "-nVersion: 
stmp += mySessionInfo.getProperty( "Version" )

sTmp  = "-nHost: 
stmp += mySessionInfo.getProperty( "host" )

stmp += "-nport: 
stmp += mySessionInfo.getProperty( "port" )

stmp += "-nciphersuite: 
stmp += mySessionInfo.getProperty( "ciphersuite" )

stmp += "-nCertificateLabel: 
stmp += mySessionInfo.getProperty( "certificateLabel" )

stmp += "-nUserData: 
stmp += mySessionInfo.getProperty( "UserData" )

stmp += "-ntokenName: 
stmp += mySessionInfo.getProperty( "tokenName" )

stmp += "-npkcs11Module: 
stmp += mySessionInfo.getProperty( "pkcs11Module" )

stmp += "-nPlease enter your choice: 
stmp += "-n 1: Accept this connection"
stmp += "-n 2: Reject this connection"
stmp += "-n 3: Accept this connection and mark CA as
trusted"
stmp += "-n 4: Reject this connection and mark CA as
untrusted"
stmp += "-n 5: Accept this CA throughout this session"
stmp += "-n 6: Reject this CA throughout this session"
// Display information in a response window and return
openwithparm(w_response, stmp)
lst_rc = Message.StringParm
return long(lst_rc)
CHAPTER 10  PowerScript Functions

See also
 ConnectToServer
 GetCertificateLabel
 GetCredentialAttribute
 GetPin

**TypeOf**

**Description**
 Determines the type of an object or control, reported as a value of the Object enumerated datatype.

**Applies to**
 Any object

**Syntax**

```
objectname.TypeOf()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectname</td>
<td>The name of the object or control for which you want the type</td>
</tr>
</tbody>
</table>

**Return value**
 Object enumerated datatype. Returns the type of objectname. If objectname is null, TypeOf returns null.

**Usage**
 Use TypeOf to determine the type of a selected or dragged control.

**Examples**

If `dw_Customer` is a DataWindow control, this statement returns DataWindow!:

```
dw_Customer.TypeOf()
```

This example looks at the first five controls in the `w_dept` window’s Control array property. The loop executes some statements for each control that is a CheckBox:

```
integer n
FOR n = 1 to 5
    IF w_dept.Control[n].TypeOf() = CheckBox! THEN
        ... // Some processing
    END IF
NEXT
```

This loop stores in the `winobjecttype` array the type of each object in the window’s Control array property:

```
object winobjecttype[]
long ll_count
```
FOR ll_count = 1 to UpperBound(Control[]) winobjecttype[ll_count] = &
   TypeOf(Control[ll_count])
NEXT

If you 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

Syntax  
   menuname.Uncheck ()
CHAPTER 10  PowerScript Functions

Return value

Integer. Returns 1 if it succeeds and -1 if an error occurs. If `menuname` is null, `Uncheck` returns null.

Usage

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

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

**Equivalent syntax**

You can set the object’s Checked property instead of calling `Uncheck`:

```power
menuname.Checked = false
```

This statement:

```power
m_appl.m_view.m_grid.Checked = FALSE
```

is equivalent to:

```power
m_appl.m_view.m_grid.Uncheck()
```

**Examples**

This statement removes the checkmark next to the `m_grid` menu selection in the drop-down menu `m_view` on the menu bar `m_appl`:

```power
m_appl.m_view.m_grid.Uncheck()
```

This example checks whether the `m_grid` menu selection in the drop-down menu `m_view` of the menu bar `m_appl` is currently checked. If so, the script unchecks the item. If it is not checked, the script checks the item:

```power
IF m_appl.m_view.m_grid.Checked = TRUE THEN
  m_appl.m_view.m_grid.Uncheck()
ELSE
  m_appl.m_view.m_grid.Check()
END IF
```

**See also**

`Check`
# Undo

**Description**
Cancels the last edit in an edit control, restoring the text to the content before the last change.

**Applies to**
DataWindow, MultiLineEdit, RichTextEdit, and SingleLineEdit controls

**Syntax**

```
editname.Undo()
```

**Argument**

<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, 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.</td>
</tr>
</tbody>
</table>

**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 )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>units</td>
<td>An integer whose value is the number of PowerBuilder units you want to convert to pixels</td>
</tr>
</tbody>
</table>
| 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:

```
integer Value
Value = UnitsToPixels(350, YUnitsToPixels!)
```

**See also**
PixelsToUnits

### 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>
</table>
| objectref | The name of the OLE control or the fully qualified name of a OLE DWObject within a DataWindow control that contains the object for which you want to establish a link.  
  The fully qualified name for a DWObject has this syntax:
  
  `dwcontrol-object.dwobjectname` |
### UpdateLinksDialog

<table>
<thead>
<tr>
<th><strong>Return value</strong></th>
<th>Integer. Returns 0 if it succeeds and -1 if an error occurs.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage</strong></td>
<td>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). <strong>UpdateLinksDialog</strong> and <strong>LinkTo</strong> are useful when a linked file has been moved and the container’s LinkUpdateOptions property is set for manual update. <strong>UpdateLinksDialog</strong> 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. <strong>LinkTo</strong> If you want to establish a link without involving the user, call the <strong>LinkTo</strong> 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 <strong>LinkTo</strong> function. If the OLE container holds an embedded object, calling <strong>UpdateLinksDialog</strong> has no effect. It returns zero because no link is broken. For more information about updating links, see <em>Application Techniques</em>.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>This example looks for the linked file for an OLE control <code>ole_report</code>. If the file is missing, it prompts the user to display the Links dialog and edit the link: <code>ole_report.UpdateLinksDialog()</code> This example looks for the linked file for an OLE DObject <code>ole_word</code> in the DataWindow control <code>dw_customer_data</code>. If the file is missing, the user can choose to edit the link using the Links dialog: <code>dw_customer_data.Object.ole_word.UpdateLinksDialog()</code></td>
</tr>
<tr>
<td><strong>See also</strong></td>
<td>InsertObject LinkTo</td>
</tr>
</tbody>
</table>

---

1114 PowerBuilder
**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

**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:

```powershell
integer a[5]
UpperBound(a)  // Returns 5
UpperBound(a,1) // Returns 5
UpperBound(a,2) // Returns -1; no 2nd dimension

integer b[10,20]
UpperBound(b,1) // Returns 10
UpperBound(b,2) // Returns 20

integer c[]
UpperBound(c)  // Returns 0; no memory allocated
  c[50] = 900
UpperBound(c)  // Returns 50
  c[60] = 800
UpperBound(c)  // Returns 60
  c[60] = 800
  c[50] = 700
UpperBound(c)  // Returns 60

integer d[10 to 50]
UpperBound(d)  // Returns 50
```

This example determines the position of a menu bar item called File, and if the item has a cascading menu with an item called Update, disables the Update item. The code could be a script for a control in a window.

The code includes a rather complicated construct: Parent.Menuid.Item. Its components are:

- **Parent** – The parent window of the control that is running the script.
- **Menuid** – A property of a window whose value identifies the menu associated with the window.
- **Item** – A property of a menu that is an array of items in that menu. If Item is itself a drop-down or cascading menu, it has its own item array, which can be a fourth qualifier.

The script is:

```powershell
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.
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:

```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 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
Capitalize 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:

```powerbuilder
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:

```powerbuilder
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:

```powerbuilder
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()

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:

integer Width
Width = w_employee.WorkSpaceWidth()

See also
PointerX
PointerY
WorkSpaceHeight
WorkSpaceX
WorkSpaceY
**WorkSpaceX**

**Description**
Obtains the distance between the left edge of a window’s workspace and the left edge of the screen.

For custom MDI frames, WorkSpaceX obtains the distance between the left edge of the frame window and the left side of the workspace area.

**Applies to**
Window objects

**Syntax**

\[ \text{windowname}.\text{WorkSpaceX}(\ ) \]

**Argument** | **Description**  
--- | ---  
\text{windowname} | The name of the window for which you want the distance between the left edge of the workspace area and the left edge of the screen.

**Return value**
Integer. Returns the distance that the left edge of the workspace area of \text{windowname} is from the left edge of the screen (in PowerBuilder units). \text{WorkSpaceX} returns -1 if an error occurs. If \text{windowname} is null, \text{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 w_employee window:

```powershell
integer workx  
workx = w_employee.WorkSpaceX()
```

**See also**
PointerX
PointerY
WorkSpaceHeight
WorkSpaceWidth
WorkSpaceY
### WorkSpaceY

**Description**

Obtains the distance between the top of a window’s workspace and the top of the screen.

For custom MDI frames, `WorkSpaceY` obtains the distance from the top of the frame window and the top of the workspace area. The top of the frame window is the lower edge of the menu bar or toolbar, if any.

**Applies to**

Window objects

**Syntax**

```
windowname.WorkSpaceY()
```

**Argument** | **Description**
--- | ---
`windowname` | The name of the window for which you want the distance between the top of the workspace area and the top of the screen.

**Return value**

Integer. Returns the distance that the top of the workspace area of `windowname` is from the top of the screen (in PowerBuilder units). If an error occurs, `WorkSpaceY` returns -1. If `windowname` is null, `WorkSpaceY` returns null.

**Usage**

The workspace area is the area between the sides of the window (not including the thickness of the frame or the vertical 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 top of the screen to the top of the workspace area in the `w_employee` window:

```
integer worky
worky = w_employee.WorkSpaceY()
```

**See also**

- PointerX
- PointerY
- WorkSpaceHeight
- WorkSpaceWidth
- WorkSpaceX
Write

Description
Writes data to an opened OLE stream object.

Applies to
OLEStream objects

Syntax

\texttt{olestream:Write ( dataforstream )}

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 \texttt{null}, Write returns \texttt{null}.

Examples
This example opens an OLE object in the file \texttt{MYSTUFF.OLE} and assigns it to the OLEStorage object \texttt{olest_stuff}. Then it opens the stream called info in \texttt{olest_stuff} and assigns it to the stream object \texttt{olestr_info}. It writes the contents of the blob variable \texttt{lb_info} to the stream \texttt{olestr_info}. Finally, it saves the storage \texttt{olest_stuff}:

```powerbuilder
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

1124 PowerBuilder
CHAPTER 10  PowerScript Functions

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 }))

Argument Description

<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>
<tr>
<td>validationscheme (optional)</td>
<td>A value of the ValSchemeType enumerated datatype specifying the validation method used by the SAX parser. Values are:</td>
</tr>
<tr>
<td></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 (optional)</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>namespaceprocessing (optional)</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 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.</td>
</tr>
<tr>
<td>schemaprocessing (optional)</td>
<td>A boolean specifying whether schema support is enabled. When set to false, the parser does not process any schema found. If schemaprocessing is true, namespaceprocessing must also be set to true. The default is false.</td>
</tr>
<tr>
<td>schemafullchecking (optional)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### XMLParseFile

**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:

```
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:

```
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:

```
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:

```
long ll_ret
ll_ret = XMLParseFile("c:\mydoc.xsd")
```

See also

ImportFile
XMLParseString
ImportFile in the DataWindow Reference or online Help
**XMLParseString**

**Description**

Parses an XML string and determines whether the string is well formed or complies with a specified grammar.

**Syntax**

```powerbuilder
XMLParseString (xmlstring {, validationscheme } {, parsingerrors } {, namespaceprocessing {, schemaprocessing {, schemafullchecking }}})
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmlstring</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:  
| (optional)        | • 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. |
| (optional)        |                                                                            |
| 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.  
| (optional)        | 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 false, the parser does not process any schema found.  
| (optional)        | 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.  
| (optional)        | Setting schemafullchecking to true has no effect unless schemaprocessing is also set to true.  
|                   | The default is false.                                                     |
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.
XMLParseString

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:

```cpp
// 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:

```cpp
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:

```cpp
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**

```
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:

```
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
Index

-- (assignment shortcut) 120
- see dashes

Symbols

! (enumerated value) 28
& see ampersand
* (multiplication) 68
+ (addition) 68
++ += (assignment shortcuts) 120
/ (division) 68
// (comments) 4
/(assignment shortcut) 120
:S (SQL) 15
< (less than) 70
<= (less than or equal) 70
= (assignment) 38
= (relational) 70
> (greater than) 70
>= (greater than or equal) 70
? (dynamic SQL) 170, 171, 174
^ (exponentiation) 68
_Is_A function 642
_Narrow function 736
~ see tilde
' see quotes

A

Abs function 306
absolute value 306
access levels
functions 58
group label 44
variables 41
ACos function 306
Activate event 182
Activate function 307
active sheet 760
active window 800
Adaptive Server Enterprise 1072
AddCategory function 309
AddColumn function 310
AddData function 311
AddItem function 313
addition operator 68
AddLargePicture function 318
AddPicture function 319
address keyword 1100
address, mail 699, 709, 710
AddSeries function 320
AddSmallPicture function 321
AddStatePicture function 322
AddToList function 323
AllowEdit property 934
ampersand (&) 15
ancestor
calling function or event 112
hierarchy 349
objects 82
return values from events 113
script, calling 121
AncestorReturnValue variable 113
AND operator 70
angle
calculating arccosine 306
calculating arcsine 328
calculating arctangent 329
calculating cosine 383
calculating sine 1041
calculating tangent 1077
converting to/from radians 796
animation
starting 797
stopping 1058
ANSI, string conversion 481, 1083, 1088
Any datat-type 24
API and database handles 404
Index

application
  closing DDE channel 359
  connecting to 369, 371, 372, 375
  elapsed time 384
  exporting object as syntax 677
  handle 485, 585
  listing objects 673
  posting messages 810
  recreating objects from syntax 678
  restarting 884
  retrieving arguments 366
  running 896
  server 1054, 1060
  terminating 135
  yielding to 1132
application name 1053, 1054, 1060
Application objects, SetTransPool function 1026
Arabic functions
  IsAllArabic 644
  IsAnyArabic 645
  IsArabic 646
  IsArabicAndNumbers 647
arccosine 306
arcsine 328
arctangent 329
arguments
  command line 366
  for events 180
  functions and events 104
  hot link 1053, 1059
  server application 1054, 1060
arithmetic operators 68
Arrange function 324
ArrangeOpen enumerated datatype 760
ArrangeSheets function 325
ArrangeTypes enumerated datatype 325
array functions
  LowerBound 697
  UpperBound 1115
arraylists 54
arrays
  about 46
  assigning values 52, 54, 118
  chars and strings 76
  copying 118
  default values 49
errors 55
example 327
initializing 54
input parameter for dynamic SQL 974
mailRecipient 699
message ID 701
passing as arguments 106
stream 857, 1124
variable-size 50
arrow pointer 992
Asc function 326
AscA function 326
ASCII values
  converting characters to 326
  of nonprinting characters 840
ASin function 328
assignment
  arrays 49, 52, 54
  overflow 75
  shortcut operators 120
  statements 118
asterisk in text patterns 718
ATan function 329
AttachmentFile property 707
audio (beep) 329
AutoCommit 1072
Autoinstantiate setting 83
animation 953, 954, 955
axis, graphs
  categories 309, 340, 406, 612
  inserting data 616
B
back quote 121
background color, graphs
  data points 511, 967
  series 565, 1003
background layer of DataWindow 994
backslash in text patterns 717
backspace, specifying 7
bands, DataWindow, moving objects to 994
BAT file 897
batch applications 811
beam pointer 992
Index

Beep function 329
BeginDrag event 183
BeginLabelEdit event 186
BeginRightDrag event 187
BeginTransaction function 330
birth dates 1131
bitmaps
  assigning to picture control 991
  in rich text 634
  printing 823
  retrieving from clipboard 353
blob datatype 19
Blob function 331
blob functions
  Blob 331
  BlobEdit 332
  BlobMid 334
  Len 668, 670
BlobEdit function 332
BlobMid function 334
blobs
  assigning to picture control 991
  converting to bytes 338, 488
  converting to string 331, 1061
  declaring 35
  extracting values from 395, 399, 405, 636, 689, 691, 858, 1079
  inserting data into 332
  reading streams into 857
  selecting from database 160
  updating 163
  writing to stream 1124
boolean datatype 19
border
  determining distance from 798, 799
  printing 834, 837, 838
bottom layer of DataWindow 994
bound 697, 1115
brackets in text patterns 717
BuildModel function 335
byte datatype 20
Byte function 338

C
C functions
  decoding returned values 639
  passing values to 689
CALL statement
  about 121
  not using 180
Cancel button 724
Cancel function 339
cancellation
  allowing 1132
  of edits 1112
  of pipeline object 339
  of printing 824
CanUndo function 339
capitalization
  in category names 309, 612
  in series names 320
  lowercase 697
  uppercase 1115
caret in text patterns 717
carriage return
  in INI files 852
  specifying 7
cascaded windows, arranging sheets 325
cascading opened windows 760
case sensitivity, comparisons 70
categories, graphs
  adding data values to series 309, 311
  adding to a series 309
  clicked 739
  counting 340
  deleting 406, 875
  identifying 340, 341
  importing data 591, 594, 598
  InsertCategory function 309
  inserting 612
  new 309
CategoryCount function 340
DirectoryName function 341
Ceiling function 342
century 1131
ChangeDirectory function 342
ChangeMenu function 343
channel, DDE 359, 758
Index

char datatype
   about 20
   array 76
      converting to string 76
CharA function 345
character array 1124
characters
   array 857
      changing capitalization 697, 1115
      converting to ASCII values 326
      extracting 344, 345, 726
      mask 984
      matching 716
      returning rightmost 889, 890
      selected 926, 929
      selecting 937
CharW function 344
Check function 346
Checked property 1110
child windows
   obtaining parent 790
   opening 745, 782
CHOOSE CASE statement 122
ChooseColor function 347
class
   contrasted with object 78
      of object 349
      OLE 613
class hierarchy 27
class user objects 80
ClassDefinition objects, FindMatchingFunction 475
ClassList function 347
ClassName function 349
Clear function 351
ClearBoldDates function 352
clearing text 351
Clicked event 190, 740
clipboard
   contents as replacement text 873
      copying 380
      cutting 390
      importing data from 591
      pasting and linking 793
      pasting from 791
      retrieving and replacing contents 353
Clipboard function 353
CLOSE Cursor statement 147
Close event 196, 355, 884
Close function 355
CLOSE Procedure statement 148
CloseChannel function 359
CloseQuery event 198, 356
CloseTab function 360
CloseUp event 199
CloseUserObject function 361
CloseWithReturn function 362
closing
   DDE channel 359
   print job 826
   windows 356
code
   generating DataWindow 1071
   object 677
   reusing 812
cold link 435, 559, 758, 1000
CollapseItem function 365
colors
   and edit masks 984
   data point 511, 878, 967
   red, green, and blue components of 887
   series 564, 1003
   supported 527
   table of standard colors 888
ColumnClick event 200
columns
   determining insertion point position 804
   in list 614
      pasting text into 792
COM file 897
command line, retrieving arguments 366
CommandParm function 366
commands
   getting from DDE client 494
   receiving form DDE application 883
comments
   in library 672
   using 4
COMMIT statement 149
CommitTransaction function 367
comparing
   numbers 636, 720, 729
   comparing strings 70
computer
  beeping 329
  reporting CPU time 384
concatenation operator 71
configuration settings
  reading 850, 852
  saving 996
CONNECT statement 150
Connection objects
  ConnectToServer function 377
  CreateInstance function 387
  DisconnectServer function 424
connections, to OLE object 369, 371
ConnectToNewObject function 369
ConnectToNewRemoteObject function 371
ConnectToServer function 377
constants
  assigning values 38
  declaring 45
  where to declare 32
Constructor event 201
ContextInformation objects
  GetCompanyName function 496
  GetFixesVersion function 536
  GetHostObject function 540
  GetMajorVersion function 547
  GetMinorVersion function 549
  GetName function 550
  GetShortName function 570
  GetVersionName function 584
ContextKeyword objects, GetContextKeywords
  function 497
context-sensitive Help 1036
continuation character 15
CONTINUE statement 124
continuous line style
  setting for data points 969
  setting for series 1005
Control array 774, 776
control structures
  CHOOSE CASE 122
  DO...LOOP 129
  FOR...NEXT 132
  IF...THEN 136
determining type 1109
dragging 427
focus of 537, 976
hiding 587, 733
moving 733
obtaining handle 585
redrawing 999
referencing 364
resizing 880
yielding 1132
Copy function 380
copying
  importing from clipboard 591
  to clipboard 380
CopyRTF function 381
CORBA Current, initializing 602
Cos function 383
cosine 383
count, of data points in a series 392
CPU
  getting information about 527
  time 384
Cpu function 384
CREATE statement 125, 800
CreateDirectory function 384
CreateInstance function 387
CreatePage function 389
cross mouse pointer 992
crosstabs, creating from source code 1071
current
  row and scrolling 916, 918
  sheet 760
cursor
  custom 992
displaying pop-up menus 800
  print 819
cursors, database
  closing 147
  declaring 146, 151
  opening 158
Index

custom class user objects 83
Cut function 390
cutting, to clipboard 390

d
dash line style
  about 969, 1005
  setting for series 1005
dashes, prohibiting in variable names 5
DashesInIdentifiers option 5
data
  adding to a graph series 311, 313
  clearing 874
  converting to type long 688, 690
  correcting pipeline 869
  finding in DataWindow 464
  from OLE server 505
  getting DDE 507
  importing 591
  inserting into a blob 332
  obtaining from control 502
  receiving from DDE application 883
  sending to DDE client 964
  sharing 393
  to OLE server 963
  transferring 1046
  writing to file 459, 461
  writing to stream 1124
data expressions, Any datatype 26
Data Pipeline painter 339, 1047
data points
  adding to a scatter graph 313
  clicked 739
  deleting 406
  inserting 615
  reporting appearance of 510
  reporting explosion percent 509
  resetting colors 878
  setting style 967
  value of 502, 516
data types
  literals 20, 21
  database stored procedures 143

databases
  canceling changes 158
  committing changes 149
  connecting to 150
  cursor, opening 158
  deleting rows 153, 154
  disconnecting from 154
  fetching rows 156
  handle 404
  inserting rows 157
  on restart 884
  repairing 869
  selecting rows 159
  transactions 1026
  transferring data between 1046
  updating 162
  updating cursored row 164
DataChange event 202
DataSource function 393
datatype checking and conversion functions
  Asc 327
  Char 344, 345
  Date 395
  DateTime 398
  Dec 405
  Double 425
  Integer 636
  IsDate 650
  IsNull 654
  IsNumber 655
  IsTime 658
  Long 688, 690
  Real 858
  String 1060
  Time 1079
datatype mappings, EAServer 29
datatyps
  about 19
  assignment 75
  blob 331
  byte 338
  date 398
determining 349
  effect of operators 73
  enumerated 28
  external functions 60
Index

literals 20, 21, 22, 24, 74
mismatch when pasting 792
numeric 73
promotion 73
promotion for function arguments 102
real 858
setting to NULL 987
standard 19
string 1060
system object 27
time 1079
unknown 24
windows 743

DataWindow control
data expressions and Any datatype 26
for pipeline errors 1047

DataWindow functions
CanUndo 339
CategoryCount 340
CategoryName 341
Clear 351
Clipboard 353
Copy 380
Cut 390
DataCount 392
FindCategory 466
FindNext 477
FindSeries 478
GetData 502
GetDataPieExplode 509
GetDataStyle 510
GetSeriesStyle 564
LineCount 679
ObjectAtPointer 739
Paste 791
PasteRTF 794
Position 805
ReplaceText 873
ResetDataColors 878
Scroll 914
SelectedLength 926
SelectedLine 927
SelectedStart 929
SelectedText 930
SelectText 937
SeriesCount 947

SeriesName 948
SetDataPieExplode 965
SetDataStyle 967
SetPosition 994
SetSeriesStyle 1003
TextLine 1078
Undo 1112

DataWindow object
creating from SELECT statement 1071
deleting from libraries 673
exporting as syntax 677
listing 673
recreating from syntax 678
date datatype 20
Date function 395
date, day, and time functions
Day 400
DayName 401
DayNumber 402
DaysAfter 403
Hour 588
Minute 730
Month 733
Now 738
RelativeDate 866
RelativeTime 866
Second 920
SecondsAfter 920
Today 1084
Year 1131

DateChanged event 202
dates
checking string 650
converting to 396
DateTime datatype 395, 399
day of week 401, 402
determining interval 403
getting dynamic 520, 522
in blobs 395
obtaining current 1084
obtaining day of month 400

DateSelected event 203
DateTime datatype 20
DateTime function 398
Day function 400
DayName function 401
Index

DayNumber function 402
DaysAfter function 403
dBase file, importing data from 594, 598
DBHandle function 404
dbsign 146
DDE channel
  closing 359
  requesting data 560
DDE client functions
  CloseChannel 359
  ExecRemote 435
  GetDataDDE 507
  GetDataDDEOrigin 507
  GetRemote 559
  OpenChannel 758
  RespondRemote 883
  SetRemote 1000
  StartHotLink 1053
  StopHotLink 1059
DDE server functions
  GetCommandDDE 494
  GetCommandDDEOrigin 495
  GetDataDDE 507
  GetDataDDEOrigin 507
  RespondRemote 883
  SetDataDDE 964
  StartServerDDE 1054
  StopServerDDE 1060
DDL, executing through dynamic SQL 168, 169
Deactivate event 204
DebugBreak function 405
Dec function 405
decimal datatype
  about 21
  converting to 405
  declaring 35
declarations
  access levels 41
  arrays 46
  constants 45
  expressions as initial values 39
  external functions 55
  syntax 35
  variables 31
  where to declare 32
DECLARE Cursor statement 151
DECLARE Procedure statement 151
definition, font for printing 828
DELETE statement 153
DELETE Where Current of Cursor statement 154
DeleteAllItems event 204
DeleteCategory function 406
DeleteColumn function 407
DeleteColumns function 407
DeleteData function 408
DeleteItem event 205
DeleteItem function 408
DeleteLargePicture function 412
DeleteLargePictures function 412
DeletePicture function 413
DeletePictures function 413
DeleteSeries function 414
DeleteSmallPicture function 415
DeleteSmallPictures function 415
DeleteStatePicture function 416
DeleteStatePictures function 416
descendant
  determining class of 349
  opening user object 767, 768, 775, 776
  opening window 746
  return values from events 113
DESTROY statement
  about 128
  ending a mail session 704
DestroyModel function 417
Destructor event 206, 360, 361
detail bands, moving objects to 994
diagonal fill pattern 971, 1006
dialog
  Insert Object 633
  Open File 528
  PasteSpecial 795
  Save File 533
diamond fill pattern 971, 1007
dimension 697
dimension of array 1115
directory, of library 673, 675
DirectoryExists function 417
DirList function 418
DirSelect function 420
Disable function 421
DisableCommit function 422
DISCONNECT statement 154
DisconnectObject function 423
DisconnectServer function 424
display format, applying to string 1060
distributed applications
  ConnectToServer function 377
  DisconnectServer function 424
  SharedObjectDirectory function 1029
  SharedObjectGet function 1030
  SharedObjectRegister function 1033, 1034
division 730
division operator 68, 69
DLL files, executing functions from 62
DLLs for external functions 58
document windows 760
dollar sign in text patterns 717
dot notation
  about 34
  instance variables 33
  structures 78
dotted line style
  setting for data points 969
  setting for series 1005
double colon 121
double datatype 21
Double function 425
DoubleClicked event 207
DoubleParm property 763, 770, 772, 778, 781
DoVerb function 426
Drag function 427
DragDrop event 210
DragEnter event 214
DraggedObject function 428
dragging, TreeView items 973
DragLeave event 215
DragObject functions
  ClassName 349
  Drag 427
  Hide 587
  Move 734
  PointerX 798
  PointerY 799
  PostEvent 811
  Print 817
  Resize 880
  SetFocus 976
  SetPosition 993
  SetRedraw 999
  Show 1034
  TriggerEvent 1100
  TypeOf 1109
DragWithin event 217
Draw function 430
drawing objects
  and SetFocus function 976
  posting events 811
  setting color of 888
DrawObject functions
  ClassName 349
  Hide 587
  Move 734
  Print 817
  Resize 880
  Show 1034
  TypeOf 1109
DropDown event 219
DropDownListBox control, deleting text 351
DropDownListBox functions
  AddItem 314
  Clear 351
  Copy 380
  Cut 390
  DeleteItem 409
  DirList 418
  DirSelect 420
  DraggedObject 428
  FindItem 469
  InsertItem 620
  Paste 791
  Position 805
  Post 810
  ReplaceText 873
  Reset 874
  SelectedLength 926
  SelectedStart 929
  SelectedText 930
  SelectItem 932
  SelectText 937
  Text 1077
  TotalItems 1086
DropDownPictureListBox functions
  AddItem 315
Index

AddPicture 319
Clear 351
Copy 380
Cut 390
DeletePicture 413
DeletePictures 413
FindItem 469
InsertItem 621
Paste 791
Position 805
ReplaceText 873
SelectedLength 926
SelectedStart 929
SelectedText 930
SelectItem 932
SelectText 937
Text 1077
TotalItems 1086

DWOObjects, OLE functions 307, 380, 426, 1113
dynamic calls
about 97
errors 99
dynamic libraries 323, 982
dynamic library (DLL) 1053
dynamic SQL
about 165
considerations 167
DynamicDescriptionArea 166
DynamicStagingArea 166
Format 1 168
Format 2 169
Format 3 170
Format 4 173
formats listed 165
NULL values 170, 171
ordering statements 167
preparing DynamicStagingArea 167
statements 166
dynamic SQL functions
GetDynamicDate 520
GetDynamicDateTime 522
GetDynamicNumber 524
GetDynamicString 525
GetDynamicTime 526
SetDynamicParm 974

DynamicDescriptionArea
about 166
properties 174
DynamicStagingArea
about 166
preparing 167

E
EAserver datatype mappings 29
edit control
counting lines in 679
deleting text from 351
determining insertion point position 804
inserting clipboard contents 353
replacing text 873
selected text 926, 929
EditLabel function 385, 431
EditMask functions
CanUndo 339
Clear 351
Copy 380
Cut 390
GetData 504
LineCount 679
LineLength 680
Paste 791
Position 805
ReplaceText 873
Scroll 914
SelectedLength 926
SelectedLine 927
SelectedStart 929
SelectedText 930
SelectText 937
SetMask 984
TextLine 1078
Undo 1112
embedded SQL 143
Enable function 432
EnableCommit function 433
Enabled property 587, 999
EndLabelEdit event 220
EntryList function 434
enumerated datatypes 28
Index

FileRead function 452
FileReadEx function 454
files
   importing data from 594
   linking 683
   security and sharing violation 445, 446
FileSeek function 456, 457
FileChooser function 458
FileWrite function 459
FileWriteEx function 461
Fill function
   about 463
   and printing 463
FillA function 464
FillPattern 513, 970, 1006
FillW function 463, 464
filtering filenames 528, 533
Find function 464
FindCategory function 466
FindClassDefinition function 467
FindFunctionDefinition function 468
FindItem function 469
FindMatchingFunction function 475
FindNext function 477
FindSeries function 478
FindTypeDefinition function 479
flicker 999
focus
   and line length 680
   finding control with 537
   selected text 927, 930, 931, 938
   setting 976
folder 673
fonts
   and string length when printing 848
   defining for printing 828
   FontFamily enumerated datatype 828
   FontPitch enumerated datatype 828
   names and sizes 829
   setting 842
   when printing 819
   when printing DataWindow controls 827
footer, moving objects to 994
foreground color
   data points 511, 967
   series 565, 1003
foreground layer of DataWindow 994
Form presentation style 1071
formats, applying to strings 1061
formfeed, specifying 7
frame window 800, 1122, 1123
FromAnsi function 481
FromUnicode function 481
function object
   exporting as syntax 677
   listing 673
   re-creating from syntax 678
functions
   about 89
   access level for external 58
   ancestor 112
   arguments 104
   calling global and system 110
   cascaded calls 108, 110
   case sensitivity 110
   chars as arguments 76
   DLLs 58
   errors when calling 99
   external 55
   external datatypes 60
   external, defined 90
   external, mail 703
   external, reporting database handle 404
   finding 92
   overloading 102
   overriding 102
   posting 94, 108
   return values 107
   similarities to events 91
   static and dynamic 96
   system, defined 90
   triggering 94
   type promotion 102
   user-defined 90

garbage collection 82, 126, 128
GarbageCollect function 482
GarbageCollectGetTimeLimit function 482
GarbageCollectSetTimeLimit function 483
Gesture event 229, 276
Index

GetActiveSheet function 484
GetAlignment function 485
GetApplication function 485
GetArgElement function 486
GetAutomationNativePointer function 487
GetByte function 488
GetCertificateLabel function 489
GetChildrenList function 491
GetColumn function 493
GetCommandDDE function 494
GetCommandDDEOrigin function 495
GetCompanyName function 496
GetContextKeywords function 497
GetContextService function 498
GetCredentialAttribute function 500
GetCurrentDirectory function 502
GetData function 502
GetDataDDE function 507
GetDataDDEOrigin function 507
GetDataPieExplode function 509
GetDataStyle function 510
GetDataValue function 516
GetDataLimits function 518
GetDisplayRange function 519
GetDynamicDate 175
GetDynamicDateTime function 520
GetDynamicDateTime function 522
GetDynamicNumber function 524
GetDynamicString 175
GetDynamicString function 525
GetDynamicTime 175
GetDynamicTime function 526
GetEnvironment function 527
GetFileOpenName function 528
GetFileSaveName function 533
GetFirstSheet function 535
GetFixesVersion function 536
GetFocus event 230
GetFocus function 537
GetFolder function 538
GetGlobalProperty function 539
GetHostObject function 540
GetItem function 541
GetItemAtPointer function 544

GetLastReturn function 545
GetLibraryList function 546
GetMajorVersion function 547
GetMinorVersion function 549
GetName function 550
GetNativePointer function 551
GetNextSheet function 552
GetOrigin function 553
GetParagraphSetting function 554
GetParent function 554
GetPin function 556
GetRecordSet function 558
GetRemote function 559
GetSelectedDate function 562
GetSelectedRange function 563
GetSeriesStyle function 564
GetShortName function 570
GetStatus function 571
GetText function 573
GetToday function 575
GetToolbar function 576
GetToolbarPos function 578, 1020
GetTransactionName function 581
GetURL function 582
GetValue function 583
GetVersionName function 584

global functions
calling 110
defined 90
global scope operator 33
global variables
about 32
scope operator 33
GOTO statement 134

Graph functions
AddCategory 309
AddData 311
AddSeries 320
CategoryCount 340
CategoryName 341
Clipboard 354
DataCount 392
DeleteCategory 406
DeleteData 408
DeleteSeries 414
FindCategory 466
<table>
<thead>
<tr>
<th>Index</th>
<th>PowerBuilder</th>
</tr>
</thead>
<tbody>
<tr>
<td>FindSeries 478</td>
<td></td>
</tr>
<tr>
<td>GetData 502</td>
<td></td>
</tr>
<tr>
<td>GetDataPieExplode 509</td>
<td></td>
</tr>
<tr>
<td>GetDataStyle 510</td>
<td></td>
</tr>
<tr>
<td>GetSeriesStyle 564</td>
<td></td>
</tr>
<tr>
<td>ImportClipboard 591</td>
<td></td>
</tr>
<tr>
<td>ImportFile 594</td>
<td></td>
</tr>
<tr>
<td>ImportString 598</td>
<td></td>
</tr>
<tr>
<td>InsertCategory 612</td>
<td></td>
</tr>
<tr>
<td>InsertData 615</td>
<td></td>
</tr>
<tr>
<td>InsertSeries 635</td>
<td></td>
</tr>
<tr>
<td>ModifyData 731</td>
<td></td>
</tr>
<tr>
<td>Reset 875</td>
<td></td>
</tr>
<tr>
<td>SaveAs 902</td>
<td></td>
</tr>
<tr>
<td>SeriesCount 947</td>
<td></td>
</tr>
<tr>
<td>SeriesName 948</td>
<td></td>
</tr>
<tr>
<td>SetDataPieExplode 965</td>
<td></td>
</tr>
<tr>
<td>SetDataStyle 967</td>
<td></td>
</tr>
<tr>
<td>SetSeriesStyle 1003</td>
<td></td>
</tr>
<tr>
<td>graphics, printing 823</td>
<td></td>
</tr>
<tr>
<td>graphs</td>
<td></td>
</tr>
<tr>
<td>categories 311</td>
<td></td>
</tr>
<tr>
<td>overlay 569</td>
<td></td>
</tr>
<tr>
<td>series 320</td>
<td></td>
</tr>
<tr>
<td>grColorType enumerated datatype 511</td>
<td></td>
</tr>
<tr>
<td>grDataType enumerated datatype 503, 516</td>
<td></td>
</tr>
<tr>
<td>Grid presentation style 1071</td>
<td></td>
</tr>
<tr>
<td>grObjectStyle enumerated datatype 739</td>
<td></td>
</tr>
<tr>
<td>Group presentation style 1071</td>
<td></td>
</tr>
<tr>
<td>grResetType enumerated datatype 875</td>
<td></td>
</tr>
<tr>
<td>grSymbolType enumerated datatype 1007</td>
<td></td>
</tr>
<tr>
<td>IsAnyHebrew 646</td>
<td></td>
</tr>
<tr>
<td>IsHebrew 650</td>
<td></td>
</tr>
<tr>
<td>IsHebrewAndNumbers 651</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td></td>
</tr>
<tr>
<td>object 880</td>
<td></td>
</tr>
<tr>
<td>workspace 1119</td>
<td></td>
</tr>
<tr>
<td>Help</td>
<td></td>
</tr>
<tr>
<td>calling Winhelp 1036</td>
<td></td>
</tr>
<tr>
<td>displaying MicroHelp 986</td>
<td></td>
</tr>
<tr>
<td>Help event 232</td>
<td></td>
</tr>
<tr>
<td>Help Search window 1036</td>
<td></td>
</tr>
<tr>
<td>hidden objects 1034</td>
<td></td>
</tr>
<tr>
<td>Hide event 233</td>
<td></td>
</tr>
<tr>
<td>Hide function 587</td>
<td></td>
</tr>
<tr>
<td>hierarchies</td>
<td></td>
</tr>
<tr>
<td>child items in a list 627, 629, 631</td>
<td></td>
</tr>
<tr>
<td>items in TreeView 365, 439</td>
<td></td>
</tr>
<tr>
<td>sorting 1043</td>
<td></td>
</tr>
<tr>
<td>sorting children 1042</td>
<td></td>
</tr>
<tr>
<td>system 27, 349</td>
<td></td>
</tr>
<tr>
<td>high word of long 639</td>
<td></td>
</tr>
<tr>
<td>highlighting</td>
<td></td>
</tr>
<tr>
<td>items in lists 932, 1055</td>
<td></td>
</tr>
<tr>
<td>scrolling 918</td>
<td></td>
</tr>
<tr>
<td>setting 1013</td>
<td></td>
</tr>
<tr>
<td>horizontal fill pattern 971, 1007</td>
<td></td>
</tr>
<tr>
<td>horizontal scrollbar for lists 314</td>
<td></td>
</tr>
<tr>
<td>horizontal scrolling, when adding items to lists 314</td>
<td></td>
</tr>
<tr>
<td>host variables in SQL 144</td>
<td></td>
</tr>
<tr>
<td>host link</td>
<td></td>
</tr>
<tr>
<td>about 964</td>
<td></td>
</tr>
<tr>
<td>determining origin of 507</td>
<td></td>
</tr>
<tr>
<td>determining source of data 508</td>
<td></td>
</tr>
<tr>
<td>establishing 1053</td>
<td></td>
</tr>
<tr>
<td>terminating 1059</td>
<td></td>
</tr>
<tr>
<td>HotLinkAlarm event 233</td>
<td></td>
</tr>
<tr>
<td>Hour function 588</td>
<td></td>
</tr>
<tr>
<td>hourglass pointer 992</td>
<td></td>
</tr>
<tr>
<td>HyperlinkToURL function 588</td>
<td></td>
</tr>
<tr>
<td>hyphens, prohibiting in variable names 5</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>icons</td>
<td></td>
</tr>
<tr>
<td>arranging in ListView 324</td>
<td></td>
</tr>
</tbody>
</table>
Index

arranging windows 325
in message box 724
identifier names, rules for 5
Idle event 234
IDs for events 179
IF...THEN statement
about 136
multiline 137
single-line 136
image
assigning to picture control 991
retrieving from clipboard 353
ImpersonateClient function 590
ImportClipboard function 591
ImportFile function 594
importing, data 594, 598
ImportString function 598
inbox
deleting messages from 701
downloading messages to 706
reading mail messages 706
retrieving message IDs from 701, 702
saving messages in 712
IncomingCallList function 600
index
highlight state of 1013, 1055
obtaining top 1085
of listbox item 924, 934
indicator variables in SQL 144
Inet objects
GetURL function 582
HyperlinkToURL function 588
PostURL function 814
Information icon 724
inheritance 82
back quote 121
double colon 121
PowerBuilder objects 27
INI file
reading 850, 852
writing values to 996
Init function 602, 603
InkEdit functions
RecognizeText 859
InkPicture functions
LoadInk 684
LoadPicture 685
ResetInk 879
ResetPicture 879
Save 898
SaveInk 911
input fields in rich text 605, 607, 608, 609, 610
InputFieldChangeData function 605
InputFieldCurrentName function 607
InputFieldDeleteCurrent function 608
InputFieldGetData function 609
InputFieldInsert function 609
InputFieldLocate function 610
InputFieldSelected event 234
Insert Object dialog 633
INSERT statement 157
InsertCategory function 612
InsertClass function 613
InsertColumn function 614
InsertData function 615
InsertFile function 619
inserting strings 870, 874
insertion point
character position 923
in editable controls 681
in text line 927, 1078
when pasting from clipboard 791
InsertItem event 235
InsertItem function 620
InsertItemFirst function 627
InsertItemLast function 629
InsertItemSort function 631
InsertObject function 633
InsertPicture function 634
InsertSeries function 635
instance variables
about 32
class of 349
dot notation 33
initialized 40
instances
checking if valid 660
defined 78
of user object 766, 769, 774, 777
Int function 636
integer
combining into long value 688, 690
Index

converting to 636
converting to char 344, 345
obtaining from blob 636
integer datatype 21
Integer function 636
Intel 527
InternetData function 638
InternetRequest objects, InternetData function 638
interpersonal messages 702
interprocess messages 702
interval 1082
IntHigh function 639
IntLow function 639
InvokePBFunction function 640
Is_A ( _Is_A ) function 642
IsAlive function 643
IsAllArabic function 644
IsAllHebrew function 644
IsAnyArabic function 645
IsAnyHebrew function 646
IsArabic function 646
IsArabicAndNumbers function 647
IsCallerInRole function 648
IsDate function 650
IsHebrew function 650
IsHebrewAndNumbers function 651
IsImpersonating function 652
IsInTransaction function 653
IsNull function 654
IsNumber function 636, 655
IsPreview function 656
IsSecurityEnabled function 656
IsTime function 658
IsTransactionAborted function 659
IsValid function 660
about 660
and Handle function 585
description 660
getting active sheet 484
getting open sheets 535, 552
ItemActivate event 235
ItemChanging event 237
ItemCollapsed event 238
ItemCollapsing event 239
ItemExpanded event 240
ItemExpanding event 241
ItemPopulate event 242
items
adding to lists 313, 620
deleting from list 409, 874
determining number of selected 1087
determining total number of 1086
highlight state of 1013, 1055
index number of 924
linking 683
selecting 932
text of 925, 1077
top 1024, 1085

J
JaguarORB, initializing 602

K
Key event 243
keyboard
determining key pressed 660
selecting text 381
KeyCode enumerated datatype
about 660
values 661
KeyDown function 661
keywords 10

L
Label presentation style 1071
labels for GOTO 6
language for OLE automation 953, 955
LastPos function 664
Layer enumerated datatype 325
Layered window 763
layering opened windows 760
layout 827
Left function 666, 667
LeftA function 667
LeftTrim function 667, 668
LeftW function 666, 667
Index

Len function 668, 670
LenA function 670
length
   line 680
   OLE stream 671
   selected text 926
   string or blob 668, 670
Length function 671
LibDirType enumerated datatype 673
LibDirType enumerated datatype 675
LibExportType enumerated datatype 677
libraries
   deleting objects from 673, 675
   pasting and linking object from 793
   search path 323, 546, 982
Library functions
   LibraryCreate 672
   LibraryDelete 673
   LibraryDirectory 673
   LibraryDirectoryEx 675
   LibraryExport 677
   LibraryImport 678
LibraryCreate function 672
LibraryDelete function 673
LibraryDirectory function 673
LibraryDirectoryEx function 675
LibraryExport function 677
LibraryImport function 678
limit, numeric 342
line spacing
   setting 844
   when printing text 819
LineCount function 679
LineDown event 244
LineLeft event 245
LineLength function 680
LineList function 681
LineRight event 246
lines
   and SetFocus function 976
   color for data points 511
   counting number of 679
   determining length 680
   graphs, color for data points 967
   graphs, color for series 565, 1003
   graphs, style for data points 512, 969
graphs, style for series 566, 567, 1005
printing 832, 846
scrolling 914
selected text 927
spacing in rich text 571
text 1078
width 513
LineUp event 247
linking
   clipboard contents 793, 795
   establishing 683
LinkTo function 683
ListBox functions
   AddItem 314
   DeleteItem 409
   DirList 418
   DirSelect 420
   FindItem 469
   InsertItem 620
   Reset 874
   SelectedIndex 924
   SelectedItem 925
   SelectItem 932
   SetState 1013
   SetTop 1024
   State 1055
   Text 1077
   Top 1085
   TotalItems 1086
   TotalSelected 1087
lists
   adding items 620
   adding new item 313
   deleting items from 874
   horizontal scrollbar 314
   of files in listbox 418
   of objects in libraries 673, 675
   sorted 314
ListView control, columns 979
ListView functions
   AddColumn 310
   AddItem 316, 317
   AddLargePicture 318
   AddSmallPicture 321
   AddStatePicture 322
   Arrange 324

PowerScript Reference Volume 2 1151
Index

DeleteColumn 407
DeleteColumns 407
DeleteItem 410
DeleteLargePicture 412
DeleteLargePictures 412
DeleteSmallPicture 415
DeleteSmallPictures 415
DeleteStatePicture 416
DeleteStatePictures 416
EditLabel 431
FindItem 470, 471
GetColumn 493
GetItem 542
GetOrigin 553
InsertColumn 614
InsertItem 622
ListView 1086
SelectedIndex 924
SetItem 978
SetOverlayPicture 988
Sort 1043
TotalItems 1086
TotalSelected 1087

literals
  datatypes of 74
  specifying 20, 21, 22, 24
LoadLink function 684
LoadPicture function 685
local variables 32
Log function
  about 687
  inverse 686
  natural logarithm 686
logarithms 686, 687
logical operators 70
LogTen function
  about 687
  inverse 687
long datatype
  about 21
  converting to 688, 690
  returning high word 639
  returning low word 639
Long function 688, 690
longlong datatype 21

LongParm
  posting events 811
  specifying values for 689
  triggering events 1100
Lookup function 692
LOOP 129
LOOP, in DO...LOOP statement 129
loops
  about 129
  iterative 132
  leaving 131
  skipping current iteration 124
  yielding within 1132
LoseFocus event 248, 725
low word of long 639
Lower function 697
LowerBound function 697
lowercase 697

M

mail functions
  mailAddress 699
  mailDeleteMessage 701
  mailGetMessages 702
  mailHandle 703
  mailLogoff 704
  mailLogon 705
  mailReadMessage 706
  mailRecipientDetails 709
  mailResolveRecipient 710
  mailReturnCode 705
  mailSaveMessage 712
  mailSend 715
mailAddress function 699
mailDeleteMessage function 701
mailHandle function 703
mailLogoff function 704
mailLogon function 705
mailLogonOption enumerated datatype 705
mailReadMessage function 706
mailReadOption enumerated datatype 707
mailRecipient structure 710
mailRecipientDetails function 709
mailResolveRecipient function 710
Index

mailReturnCode function 705
mailSaveMessage function 712
mailSend function 715
main window 734
MAPI 703
margins 819, 841, 990
masks
  applying to strings 1061
  matching 716
  reporting length of 680
  setting 984
Match function 716
Max function 720
maximum value below a limit 636
maximum value of two numbers 720
MDI Client (MDI_1) functions
  ClassName 349
  Hide 587
  Print 817
  Resize 880
  SetRedraw 999
  Show 1035
  TypeOf 1109
MDI frame
  arranging windows 325
  changing menus 343
  displaying pop-up menus 800
  getting active 484
  opening sheets 745, 760, 762
  specifying MicroHelp text 986
MDI frame functions
  ArrangeSheets 325
  GetActiveSheet 484
  GetFirstSheet 535
  GetNextSheet 552
  GetToolbar 576
  GetToolbarPos 578, 1020
  OpenSheet 760
  OpenSheetWithParm 762
  Print 817
  SetMicroHelp 986
  SetToolbar 1018
measurement 1113
member, OLE 721, 722, 723
MemberDelete function 721
MemberExists function 722
MemberRename function 723
memory
  allocation for arrays 50
  and variable-sized arrays 1115
  releasing after mail session 704
Menu functions
  Check 346
  ClassName 349
  Disable 421
  Enable 432
  PopMenu 800
  Show 1034
  TriggerEvent 1100
  TypeOf 1109
  Uncheck 1110
Menu objects
  exporting as syntax 677
  listing 673
  recreating from syntax 678
menus
  changing 343
  Checked property 346
  creating object 125
  displaying 800
  for sheet 760
message ID array 702
Message ID array
  accessing parameters 782
  and TriggerEvent function 1100
  close return value 362
  creating 125
  determining type 1110
  extracting strings from 1062, 1065
  open sheet parameters 762
  PowerObjectParm property 363
  properties 770, 772, 778, 781
  specifying values for 689
MessageBox function 724, 833
messages
  deleting 701
  posting 810
  saving 712, 715
  sending to a window 945
metacharacters 716, 717
MicroHelp 986
Index

Microsoft Windows
  and DDE 559
  and timers 1082
  calling Winhelp 1036
  defining fonts for printing 829
  displaying Save File response window 533
  events and messages in 812
  getting filenames 528
  getting information about 527
  message numbers 945
  obtaining handle 585
  returned messages 639
  RightToLeft version 644, 645, 646, 647, 650, 651, 886
Mid function 726
MidA function 728
MidW function 726, 729
Min function 729
minimum value
  above a limit 342
  of two numbers 729
Minute function 730
miscellaneous functions
  IsValid 660
  KeyDown 661
  MessageBox 796
  PixelsToUnits 796
  RGB 887
  SetNull 987
  SetPointer 992
  TypeOf 1109
  UnitsToPixels 1113
Mod function 730
Modified event 250
ModifyData function 731
modulus 527
Month function 733
month, obtaining the day of 400
More Windows menu item 761
mouse
  selecting text 381
  setting shape of pointer 992
MouseDown event 252
MouseMove event 254
MouseUp event 257
Move function 733
Moved event 260
multidimensional arrays 48, 52
MultiLineEdit functions
  CanUndo 339
  Clear 351
  Copy 380
  Cut 390
  LineCount 679
  LineLength 680
  Paste 791
  Position 805
  ReplaceText 873
  Scroll 914
  SelectedLength 926
  SelectedLine 927
  SelectedStart 929
  SelectedText 930
  SelectText 937
  TextLine 1078
  Undo 1112
multiplication operator 68, 69
MultiSelect property
  highlighted state 1013, 1059
  selecting items 925, 934

N
  names, rules for 5
  naming conventions 37
  Narrow (_Narrow) function 736
  negative numbers 1039
  nested OLE objects 751, 754
  newline, specifying 7
  NEXT, in FOR...NEXT statement 132
  NextActivity function 737
  NOT operator 70
  Now function 738
  null object references 764, 770, 772, 778, 818, 784, 786
  NULL values
    about 8
    checking 654
    dynamic SQL 171
    in boolean expressions 70
Index

setting variables to 987

testing for 8

numbers

category 341
checking string 655
comparing 720, 729
converting char 344, 345, 396, 405
determining maximum 342
determining sign of 1039
getting dynamic 524
logarithm of 686, 687
multiplying by pi 796
of day of week 402
of lines, counting 679
random 854
returning remainder 730
rounding 894
truncating 1105

numeric functions

Abs 306
ACos 306
ASin 328
ATan 329
Ceiling 342
Cos 383
Exp 438
Fact 440
Int 636
Log 686
Max 720
Min 729
Mod 730
Pi 796
Rand 854
Randomize 854
Round 894
Sign 1039
Sin 1041
Sqrt 1045
Tan 1077
Truncate 1105

N-Up presentation style 1071

O

ObjectAtPointer function 739

objects

about 78
ancestor 82
assignment 84
changing position 994
creating instance 125
deleting from libraries 673
destroying instance 128
determining class of 349
determining type 1109
garbage collection 82, 128
genral references 11
hiding 587, 734
inserting 613, 619, 633
instantiating 81
linking 683
loading 323, 982
moving 734
obtaining handle 585
parent object 554
passing as arguments 104
posting events 811
recreating 678
redrawing 999
reference handle 78
saving OLE 900
selecting 936
setting focus 976
triggering events 1100
under pointer 739

objects, Connection

ConnectToServer function 377
CreateInstance function 385, 387
DisconnectServer function 424

objects, shared

SharedObjectDirectory function 1029
SharedObjectGet function 1030
SharedObjectRegister function 1033
SharedObjectUnregister function 1034

ObjectToString function 741
OffsetPos function 742
Offsite enumerated datatype 308
OK button 724
### OLE DObject functions
- **Activate** 307
- **Copy** 380
- **DoVerb** 426
- **UpdateLinksDialog** 1113

### OLE expressions and Any datatype
26

### OLEControl functions
- **Activate** 307
- **Clear** 351
- **Copy** 380
- **Cut** 390
- **DoVerb** 426
- **GetData** 505
- **GetNativePointer** 551
- **InsertClass** 613
- **InsertFile** 619
- **InsertObject** 633
- **LinkTo** 683
- **Open** 743
- **Paste** 791
- **PasteLink** 793
- **PasteSpecial** 795
- **ReleaseAutomationPointer** 868
- **Save** 900
- **SaveAs** 905, 906
- **SelectObject** 936
- **SetAutomationLocale** 953
- **SetData** 963
- **UpdateLinksDialog** 1113

### OLECustomControl functions
- **GetData** 505
- **GetNativePointer** 551
- **ReleaseAutomationPointer** 868
- **SetAutomationLocale** 953
- **SetData** 963

### OLEObject functions
- **ConnectNewToObject** 369
- **ConnectToNewRemoteObject** 371
- **ConnectToObject** 372
- **ConnectToRemoteObject** 375
- **DisconnectObject** 423
- **GetAutomationNativePointer** 487
- **ReleaseAutomationPointer** 867
- **SetAutomationPointer** 954
- **SetAutomationTimeout** 955

### OLEStorage functions
- **Clear** 351
- **Close** 357
- **MemberDelete** 721
- **MemberExists** 722
- **MemberRename** 723
- **Open** 743
- **SaveAs** 907, 908

### OLEStream functions
- **Close** 357
- **Length** 671
- **Open** 743
- **Read** 855
- **Seek** 921
- **Write** 1124

### OPEN Cursor statement
158

### Open event
261, 884

### Open function
743

### OpenChannel function
758

### OpenSheet function
760

### OpenSheetWithParm function
762

### OpenTab function
766

### OpenTabWithParm function
769

### OpenUserObject function
774

### OpenUserObjectWithParm function
777

### OpenWithParm function
782

### operating system
information about 527

### RightToLeft version
644, 645, 646, 647, 650, 651, 886

### operators
about 67

### arithmetic
68

### assignment shortcuts
118, 120

### concatenation
71

### effect on datatypes
73

### logical
70

### precedence
72

### relational
70

### OR operator
70

### Original window
763

### Other event
264

### OutgoingCallList function
787

### oval
and SetFocus function 976

### printing
834
overflow on assignment 75
overlay 569, 1008
overloading functions 102
overriding functions 102

P
page
  printing 836
    printing borders 835, 837, 838
    size 819
PageCreated function 789
PageDown event 265
PageLeft event 266
PageRight event 267
PageUp event 268
paging functions
  ScrollNextPage 915
  ScrollPriorPage 917
paragraphs 990
parameters
  command line 366
  opening sheets with 762
  opening tab pages with 769
  opening user objects with 767, 768, 775, 776, 777
  opening windows with 782
  specifying for DynamicDescriptionArea 974
Parent pronoun 12
parent window
  changing position relative to 734
  obtaining 790
  of open window 744, 745, 782
parentheses in expressions 72
ParentWindow function 790
parsing strings 802, 804
password 705
Paste function 791
PasteLink function 793
PasteSpecial function 795
pasting
  embedding or linking 795
  from clipboard 791, 793
path
  of library file 672
OLE storage 745
returning 528
saving files 533
pattern matching 716
PBL file
  creating 672
  deleting 673
  listing contents of 673, 675
pwm_dwngraphcreate event 1004
PDB file 746
performance
  and Yield function 1132
  Any datatype 27
  dynamic function and event calls 98
  period in text patterns 717
Pi function 796
Picture functions
  ClassName 349
  Drag 427
  Draw 430
  Hide 587
  Move 734
  PointerX 798
  PointerY 799
  PostEvent 811
  Print 817
  SetFocus 976
  SetPicture 991
  SetPosition 993
  SetRedraw 999
  Show 1034
  TriggerEvent 1100
  TypeOf 1109
PictureListBox functions
  AddItem 315
  AddPicture 319
  DeletePicture 413
  DeletePictures 413
  FindItem 469
  InsertItem 621
  SelectedItem 925
  SelectItem 932
  SetTop 1024
  State 1055
  Text 1077
  Top 1085
Index

TotalItems 1086
TotalSelected 1087
pictures
  for TreeView items 981
  in listboxes 319
  in rich text 634
  in TreeView controls 319
ListView controls 318, 321, 322
overlay in lists 988
  TreeView controls 322
PictureSelected event 269
pie graphs 509, 965
PIF file 897
PipeEnd event 269
Pipeline functions
  Cancel 339
  Repair 869
  Start 1046
PipeMeter event 270
PipeStart event 271
pixels 796, 1113
PixelsToUnits function 796
Play function 797
plus sign in text patterns 718
point size 828
pointer
  determining distance from edge 798
  distance from top 799
  file 456, 457, 459, 461
  read/write 921
  returning object under 739
  setting shape 992
PointerX function 798
PointerY function 799
polymorphism for functions and events 96
PopMenu function 800
PopulateError function 801
pop-up windows
  moving 734
  obtaining parent 790
  opening 745, 782
Pos function 802, 804
PosA function 804
position
  changing 734
  of insertion point 804
  setting for control 993
Position function 804
positive numbers 1039
Post function 810
PostEvent function 811
posting functions or events 94
PostURL function 814
PowerBuilder units 796, 1113
PowerBuilder, datatypes for external functions 60
PowerObject base class 27, 79
PowerObject functions
  ClassName 349
  GetContextService 498
  GetParent 554
PowerObjectParm
  and CloseWithReturn function 363
  determining type 1110
  opening sheets with parameters 763, 770, 772, 778, 781
PowerScript statements 118
precedence of numeric datatypes 73
precedence of operators 72
presentation styles 1071
print cursor
  getting coordinates of 849
  in print jobs 819
Print function 817
print functions
  Print 817
  PrintBitmap 823
  PrintCancel 824
  PrintClose 826
  PrintDataWindow 827
  PrintDefineFont 828
  PrintOpen 833
  PrintOval 834
  PrintPage 836
  PrintRect 837
  PrintRoundRect 838
  PrintScreen 839
  PrintSend 840
  PrintSetFont 842
  PrintSetSpacing 844
  PrintSetup 845
  PrintText 846
Index

PrintWidth 848
PrintX 849
PrintY 849
print job 833
PrintBitmap function 823
PrintCancel function 824
PrintClose function 826
PrintDataWindow function 827
PrintDefineFont function 828
printer setup 840
Printer Setup dialog box 845
PrintEx function 829
PrintFooter event 271
PrintGetPrinter function 830
PrintGetPrinters function 831
PrintHeader event 272
PrintLine function 832
PrintOpen function
  about 833
  and message boxes 725
PrintOval function 834
PrintPage function 836
PrintRect function 837
PrintRoundRect function 838
PrintScreen function 839
PrintSend function 840
PrintSetFont function 842
PrintSetPrinter function 843
PrintSetSpacing function 844
PrintSetup function 845
PrintSetupPrinter function 845
PrintText function 846
PrintWidth function 848
PrintX function 849
PrintY function 849
private access
  functions 58
  variables and constants 41
PRIVATEREAD access modifier 41
PRIVATEWRITE access modifier 42
processor 527
profile files
  reading 850, 852
  writing to 995
ProfileClass objects, RoutineList function 895
ProfileInt function 850
ProfileLine objects, OutgoingCallList function 787
ProfileRoutine objects
  IncomingCallList function 600
  LineList function 681
  OutgoingCallList function 787
ProfileString function 852
Profiling functions
  BuildModel 335
  ClassList 347
  DestroyModel 417
  RoutineList 895
  SetTraceFileName 1024
  SystemRoutine 1074
pronouns
  about 11
  instance variables 34
  Parent 12
  Super 14
  This 13
properties
  and GetFocus function 537
  font, for printing 828
  getting and setting 485
  Message object 763
  setting width and height 880
  window 744, 746
property expressions, Any datatype 26
PropertyChanged event 272
PropertyRequestEdit event 273
protected access
  functions 58
  variables and constants 41
PROTECTEDREAD access modifier 41
PROTECTEDWRITE access modifier 42
public access
  functions 58
  variables and constants 41

Q
question mark
  dynamic SQL 170, 171, 174
  icon in message box 724
  in text patterns 718
  quoted strings, continuing 15
Index

quotes
   nesting 22
   rules for 23
   specifying 7
   with tilde 22

R
radians 796
Rand function 854
random numbers
   initializing generator 854
   obtaining 854
Randomize function 854
RButtonDown event 274
RButtonUp event 276
Read function 855
read-only arguments 104
real datatype 21
Real function 858
recipient, mail 709
RecognizeText function 859
rectangle
   and SetFocus function 976
   printing 837, 838
references
   and CloseWithReturn function 364
   passing arguments by 104
   passing parameters 764, 770, 772, 778, 781, 784, 786
Registration database 615
RegistryDelete function 859
RegistryGet function 860
RegistryKeys function 861
RegistrySet function 862
RegistryValues function 865
relational operators 70
RelativeDate function 866
RelativeTime function 866
ReleaseAutomationNativePointer function 867
ReleaseNativePointer function 868
remainder 730
remote DDE application 883
remote procedure calls
   declaring 63
   defined 90
   RemoteExec event 277, 494, 1055
   RemoteHotLink event 277
   RemoteHotLinkStart event 1055
   RemoteHotLinkStop event 278, 1055
   RemoteRequest event 278, 964, 1055
   RemoteSend event 279, 508, 1055
   RemoveDirectory function 868
   Rename event 279
   Repair function 869
   repairing pipeline, canceling 339
   Replace function 870
   ReplaceA function 872
   ReplaceText function 873
   ReplaceW function 874
   report view for ListView 542
   reserved words 10
   Reset function 874
   ResetArgElements function 877
   ResetDataColors function 878
   ResetInk function 879
   ResetPicture function 879
   Resize event 280
   Resize function 880
   ResolveInitialReferences function 881
   RespondRemote function 883
   response windows
      closing 362
      moving 734
   Restart function 884
   ResumeTransaction function 884
   retry button 724
   RETURN statement 138
   return values
      about 107
      event return codes 180
      from ancestor events 113
      from mail session 705
      TriggerEvent function 1100
   Reverse function 886
   RevertToSelf function 887
   RGB function 887
rich text
   alignment 485, 951
   and data 393
   copying with formatting 381, 794
   data 605, 607, 608, 609, 610
Index

determining insertion point position 806
editing header and footer 1035
find again 477
finding text 464
formatting 554, 571, 574, 990
line spacing 1012
preview 656
preview document 656, 816
printing 822, 829
save file 909
selecting 939
selecting a line 942
selecting a word 943
selecting all 941
text color 574, 1014
text settings 1014
RichTextEdit functions
CanUndo 339
Clear 351
Copy 380
CopyRTF 381
Cut 390
DataSource 393
Find 464
FindNext 477
GetAlignment 485
GetParagraphSetting 554
GetSpacing 571
GetTextColor 574
GetTextStyle 574
InputFieldChangeData 605
InputFieldCurrentName 607
InputFieldDeleteCurrent 608
InputFieldGetData 609
InputFieldInsert 609
InputFieldLocate 610
InsertPicture 634
IsPreview 656
LineCount 679
LineLength 680
Paste 791
PasteRTF 794
Position 806
Preview 816
Print 822
PrintEx 829
ReplaceText 873
SaveDocument 909
Scroll 914
ScrollNextPage 915, 916
ScrollPriorPage 917
ScrollPriorRow 918
ScrollToRow 919
SelectedColumn 923
SelectedLength 926
SelectedLine 927
SelectedPage 928
SelectedStart 929
SelectedText 930
SelectText 939
SelectTextAll 941
SelectTextLine 942
SelectTextWord 943
SetAlignment 951
SetParagraphSetting 990
SetSpacing 1012
SetTextColor 1014
SetTextStyle 1014
ShowHeadFoot 1035
Undo 1112
Right function 889
RightA 890
RightClicked event 281
RightDoubleClicked event 282
RightToLeft operating system 886
RightToLeft software 644, 645, 646, 647, 650, 651
RightTrim function 891
RightW function 890
ROLLBACK statement 158
RollbackOnly function 891
RollbackTransaction function 893
Round function 894
RoutineList function 895
rows
  correcting pipeline data 869
determining insertion point position 805
scrolling 915, 916, 918, 919
rows, database
  deleting 153, 154
  fetching 156
  inserting 157
Index

updating 162
updating cursored row 164
RPC see remote procedure calls
Run function 896

S
Save As dialog box 904
Save event 284
Save File response window 533
Save function 900
Save function (InkPicture controls) 898
SaveDocument function 909
SaveInk function 911
SaveObject event 285
scatter graphs
adding values to series 313
changing data point values 732
importing data 591, 594, 596, 598
inserting data from strings 599
obtaining data point values 503
scope operator 110
screen
changing position relative to 734
display 527
distance to workspace 1122, 1123
printing 839
scripts
stopping execution 884
terminating 138
triggering events 1100
Scroll function 914
ScrollHorizontal event 725
scrolling
ListBox 1024
TreeView 975
scrolling functions
Scroll 914
ScrollNextPage 915
ScrollNextRow 916
ScrollPriorPage 917
ScrollPriorRow 918
ScrollToRow 919
Top 1085
ScrollNextPage function 915
ScrollNextRow function 916
ScrollPriorPage function 917
ScrollPriorRow function 918
ScrollToRow function 919
ScrollVertical event 725
searching, rich text 464, 477
Second function 920
SecondsAfter function 920
Seek function 921, 923
SeekType enumerated datatype 921
SELECT statement 159
SELECTBLOB statement 160
Selected event 286, 986
SelectedColumn function 923
SelectedIndex function 924
SelectedItem function 925
SelectedLength function 926
SelectedLine function 927
SelectedPage function 928
SelectedStart function 929
SelectedText function 930
selection, clearing in list 934
SelectionChanged event 287
SelectionChanging event 290
SelectionRange function 931
SelectItem function 932
SelectObject function 936
SelectText function
about 937
copying to clipboard 381
SelectTextAll function 941
SelectTextLine function 942
SelectTextWord function 943
Send function 945
sender 706
SendMessage function 945
series, graphs
adding to 320
adding values to 311
clicked 739
counting 947
data points 392, 408, 503, 516, 731, 878
deleting 414, 875
finding number of 478
importing 591, 594, 598
inserting 635
index

inserting data 615
obtaining name 948
reporting appearance of 564
setting style 1003
SeriesCount function 947
SeriesName function 948
server application
activating 308, 936
closing DDE channel 362
connecting to 369, 371, 372, 373, 375
DDE support 759
pasting and linking 793
providing data 559
sending data to 1000
sending to DDE client 964
stopping 1060
SetAbort function 949
SetAlignment function 951
SetArgElement function 952
SetAutomationPointer function 954
SetAutomationTimeout function 955
SetBoldDate function 957
SetByte function 958
SetComplete function 960
SetData function 963
SetDataDDE function 964
SetDataPieExplode function 965
SetDataStyle function 967
SetDateLimits function 972
SetDropHighlight function 973
SetDynamicParm function 974
SetFirstVisible function 975
SetFocus function 976
SetGlobalProperty function 977
SetItem function 978
SetLevelPictures function 981
SetLibraryList function 982
SetMask function 984
SetMicroHelp function 986
SetNull function 987
SetOverlayPicture function 988
SetPicture function 991
SetPointer function 992
SetPosition function 993
SetProfileString function 995
SetRange function 997
SetRecordSet function 998
SetRedraw function 999
SetRemote function 1000
SetResultSet function 1002
SetSelectedDate function 1010
SetSelectedRange function 1011
SetSeriesStyle function 1003
SetState function 1013
SetTimeout function 1015
SetToday function 1017
SetToolbar function 1018
SetTop function 1024
SetTraceFileName function 1024
SetTransPool function 1026
setup printer 840
SetValue function 1028
shade
data points 511, 967
series 565, 1003
shapes
mouse pointer 992
printing 835, 837, 838
shared objects
about 1031
SharedObjectDirectory function 1029
SharedObjectGet function 1030
SharedObjectRegister function 1033
SharedObjectUnregister function 1034
shared variables
about 32
initialized 40
SharedObjectDirectory function 1029
SharedObjectGet function 1030
SharedObjectRegister function 1033
SharedObjectUnregister function 1034
sharing data 393
sheets
arranging 325
getting active 484
getting first open 535
getting next open 552
obtaining parent 790
opening 745, 760, 762
toolbars 576, 578, 1018, 1020
Show event 292
Show function 1034
Index

ShowHeadFoot function 1035
ShowHelp function 1036
ShowPopupHelp function 1038
Sign function 1039
SignalError function 1039
signing on to database during compile, preventing 146
Sin function 1041
size 1041
SingleLineEdit functions
  CanUndo 339
  Clear 351
  Copy 380
  Cut 390
  Move 734
  Paste 791
  Position 805
  ReplaceText 873
  SelectedLength 926
  SelectedStart 929
  SelectedText 930
  SelectText 937
  Undo 1112
Sleep function 1041
Sort event 293
Sort function 1042
sort order
  and GetCalc function 585
  when inserting items into lists 621
SortAll function 1043
sounds (beep) 329
source database 1046
Space function 1045
spaces
  deleting leading 667, 668
  deleting trailing 891
  inserting in a string 1045
  removing from strings 1104
special ASCII characters in strings 6
SQL statements
  about 144
CLOSE Procedure 148
COMMIT 149
CONNECT 150
  continuing 15
DECLARE Procedure 151
DISCONNECT 154
error handling 145
EXECUTE 155, 974
FETCH 156
in pipeline execution 1047
INSERT 157
OPEN 974
OPEN Cursor 158
painting 146
ROLLBACK 158
SELECT 159
SELECTBLOB 160
UPDATE 162
UPDATE Where Current of Cursor 164
UPDATEBLOB 163
SQLCode property 145
SQLDBCode property 145
SQLErrText property 145
Sqrt function 1045
square fill pattern 971, 1007
square root 1045
Start event 295, 296
Start function
  about 1046
  canceling pipeline 339
  server application 372, 375
StartHotLink function 1053
StartServerDDE function 1054
state
  of listbox items 1055
  setting highlighted 1013
State function 1055
statements, PowerScript
  assignment 118
  CALL 121
  CHOOSE CASE 122
  CONTINUE 124
  CREATE 125
  DESTROY 128
  DO...LOOP 129
  EXIT 131

1164  PowerBuilder
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR...NEXT 132</td>
</tr>
<tr>
<td>GOTO 134</td>
</tr>
<tr>
<td>HALT 135</td>
</tr>
<tr>
<td>IF...THEN 136</td>
</tr>
<tr>
<td>listed 117</td>
</tr>
<tr>
<td>RETURN 138</td>
</tr>
<tr>
<td>separating 16</td>
</tr>
<tr>
<td>static calls 96</td>
</tr>
<tr>
<td>StaticText control, inserting clipboard 353</td>
</tr>
<tr>
<td>stgShareMode enumerated datatype 750, 753</td>
</tr>
<tr>
<td>Stop function 1057</td>
</tr>
<tr>
<td>stop sign icon 724</td>
</tr>
<tr>
<td>StopHotLink function 1059</td>
</tr>
<tr>
<td>StopServerDDE function 1060</td>
</tr>
<tr>
<td>storages, OLE</td>
</tr>
<tr>
<td>file 905</td>
</tr>
<tr>
<td>releasing 357</td>
</tr>
<tr>
<td>saving 900</td>
</tr>
<tr>
<td>stored procedures</td>
</tr>
<tr>
<td>closing 148</td>
</tr>
<tr>
<td>declaring 146, 151</td>
</tr>
<tr>
<td>executing 155</td>
</tr>
<tr>
<td>stored procedures, declaring 63</td>
</tr>
<tr>
<td>streams, OLE</td>
</tr>
<tr>
<td>checking 722</td>
</tr>
<tr>
<td>deleting 721</td>
</tr>
<tr>
<td>renaming 723</td>
</tr>
<tr>
<td>string datatype 21</td>
</tr>
<tr>
<td>String function 1060</td>
</tr>
<tr>
<td>string functions</td>
</tr>
<tr>
<td>Asc 326, 327</td>
</tr>
<tr>
<td>Char 344</td>
</tr>
<tr>
<td>CharA 345</td>
</tr>
<tr>
<td>Fill 463</td>
</tr>
<tr>
<td>FillW 463, 464</td>
</tr>
<tr>
<td>Left 666, 667</td>
</tr>
<tr>
<td>LeftTrim 667, 668</td>
</tr>
<tr>
<td>LeftW 666, 667</td>
</tr>
<tr>
<td>Len 668, 670</td>
</tr>
<tr>
<td>Lower 697</td>
</tr>
<tr>
<td>Match 716</td>
</tr>
<tr>
<td>Mid 726</td>
</tr>
<tr>
<td>MidW 726</td>
</tr>
<tr>
<td>Pos 802, 804</td>
</tr>
<tr>
<td>Replace 870, 874</td>
</tr>
<tr>
<td>Right 889</td>
</tr>
<tr>
<td>RightTrim 891</td>
</tr>
<tr>
<td>RightW 890</td>
</tr>
<tr>
<td>Space 1045</td>
</tr>
<tr>
<td>Trim 1104</td>
</tr>
<tr>
<td>Upper 1115</td>
</tr>
<tr>
<td>StringParm property 763, 770, 772, 778, 781</td>
</tr>
<tr>
<td>strings</td>
</tr>
<tr>
<td>char arrays 76</td>
</tr>
<tr>
<td>comparing 70</td>
</tr>
<tr>
<td>concatenating 71</td>
</tr>
<tr>
<td>continuing 15</td>
</tr>
<tr>
<td>converting 327, 331, 338, 396, 405, 425, 689, 691, 858</td>
</tr>
<tr>
<td>converting to char 76</td>
</tr>
<tr>
<td>deleting leading spaces 667, 668</td>
</tr>
<tr>
<td>detecting contents 650, 655, 658</td>
</tr>
<tr>
<td>determining width for printing 848</td>
</tr>
<tr>
<td>extracting 344, 345, 726</td>
</tr>
<tr>
<td>finding substrings 802, 804</td>
</tr>
<tr>
<td>getting dynamic 525</td>
</tr>
<tr>
<td>importing data from 598</td>
</tr>
<tr>
<td>lowercase 697</td>
</tr>
<tr>
<td>nested 22</td>
</tr>
<tr>
<td>uppercase 1115</td>
</tr>
<tr>
<td>writing to stream 1124</td>
</tr>
<tr>
<td>StringToObject function 1066</td>
</tr>
<tr>
<td>Stroke event 296</td>
</tr>
<tr>
<td>structure objects</td>
</tr>
<tr>
<td>exporting as syntax 677</td>
</tr>
<tr>
<td>listing 673</td>
</tr>
<tr>
<td>recreating from syntax 678</td>
</tr>
<tr>
<td>structures</td>
</tr>
<tr>
<td>about 77</td>
</tr>
<tr>
<td>assignment 84</td>
</tr>
<tr>
<td>autoinstantiated user objects 83</td>
</tr>
<tr>
<td>for return values 363</td>
</tr>
<tr>
<td>mailRecipient 710</td>
</tr>
<tr>
<td>passing as arguments 105</td>
</tr>
<tr>
<td>passing to external functions 62</td>
</tr>
<tr>
<td>passing values as 784, 786</td>
</tr>
<tr>
<td>substorages, OLE</td>
</tr>
<tr>
<td>checking 722</td>
</tr>
<tr>
<td>deleting 721</td>
</tr>
<tr>
<td>renaming 723</td>
</tr>
<tr>
<td>saving 905</td>
</tr>
</tbody>
</table>
Index

substrings
  extracting 726
  finding 802, 804
  replacing 870, 874

subtraction operator
  list of arithmetic operators 68
  surrounded by spaces 17, 68

summary, moving objects to 994

Super pronoun 14

SuspendTransaction function 1070

symbol types, graphs
  data points 513, 970
  series 1006

syntax
  exporting object as 677
  recreating objects from 678

SyntaxFromSQL function 1071

system
  date 1084
  events 179, 810
  events, defined 90
  functions 110
  object classes 79
  object datatypes 27
  object hierarchy 27
  registry 859, 860, 861, 862, 865
  time 738

system and environment functions
  Clipboard 353
  CommandParm 366
  DebugBreak 405
  FindClassDefinition 467
  FindFunctionDefinition 468
  FindTypeDefinition 479
  GarbageCollect 482
  GarbageCollectGetTimeLimit 482
  GarbageCollectSetTimeLimit 483
  GetApplication 485
  GetEnvironment 527
  Handle 585
  PopulateError 801
  Post 810
  ProfileInt 850
  ProfileString 852
  Restart 884
  Run 896
  Send 945
  SetProfileString 995
  ShowHelp 1036
  SignalError 1039
  Yield 1132

SystemError event 297
SystemKey event 298
SYSTEMREAD modifier 42
SystemRoutine function 1074
SYSTEMWRITE modifier 42

T

  tab character, specifying 7

Tab functions
  CloseTab 360
  MoveTab 735
  SelectTab 936
  TabPostEvent 1075
  TabTriggerEvent 1076

Tab pages
  changing order 735
  CreatePage function 389
  opening user objects 766, 769
  PageCreated function 789
  selecting 936

tables, database, transferring data between databases 1046

Tabular presentation style 1071

Tag property
  and GetFocus function 537
  storing MicroHelp text 986

Tan function 1077

tangent 1077

target database for pipeline 1046

temporary files 706

terminator for string 334
text
  deleting from edit controls 351
  finding in RichTextEdit 464, 477
  finding substrings 802, 804
  importing data from string 598
  line spacing when printing 819
  metacharacters 717
  MicroHelp 986
Index

obtaining current line 1077, 1078
of listbox item 925
of message box 724
on clipboard 353, 381, 390
pasting over 792
printing 818, 846
replacing 873
restoring 1112
save rich text as ASCII 909
selecting 926, 930, 937
setting color of text file
importing data from 594
saving to 902
Text function 1077
Text property 537
TextLine function 1078
This pronoun 13
tilde
in strings 22
rules for 23
specifying 7
time
checking string 658
converting to datatype 1079
CPU 384
DateTime datatype 399
getting dynamic 522, 526
minutes 730
now 738
relative 866
seconds 920
time datatype 23
Time function 1079
Timer event 299
Timer function 1082
timers, triggering event 1082
timing functions
CPU 384
Idle 589
Timer 1082
timing object
deactivating 1058
starting 1048
stopping 1058
title of message box 724
ToAnsi function 1083
Today function 1084
ToolBarMoved event 300
toolbars 576, 578, 1018, 1020
top
bringing object to 1034
determining distance from 799
moving listbox item to 1024
moving objects to 994
Top function 1085
topics
calling Help 1036
ending server application 1060
starting server application 1054
TotalColumns function 1086
TotalItems function 1086
TotalSelected function 1087
ToUnicode 1088
ToUnicode function 1088
Trace file functions, Open 743
TraceBegin function 1088
TraceClose function 1090
TraceDisableActivity function 1091
TraceEnableActivity function 1093
TraceEnd function 1095
TraceError function 1096
TraceFile objects
Close function 358
NextActivity function 737
Reset function 876
TraceOpen function 1097
TraceTree objects
BuildModel function 335
DestroyModel function 417
EntryList function 434
SetTraceFileName function 1024
TraceTreeGarbageCollect objects, GetChildrenList function 491
TraceTreeObject objects, GetChildrenList function 491
TraceTreeRoutine objects, GetChildrenList function 491
TraceUser function 1099
tracing functions
TraceBegin 1099
TraceClose 1090
Index

TraceDisableActivity 1091
TraceEnableActivity 1093
TraceEnd 1095
TraceError 1096
TraceOpen 1097
TraceUser 1099
trailer, moving objects to 994
Transaction object functions
  DBHandle 404
  SyntaxFromSQL 1071
Transaction objects, creating 125
  transparent line style, graphs
    setting for data points 969
    setting for series 1005
TreeView functions
  AddPicture 319
  CollapseItem 365
  DeleteItem 410
  DeletePicture 413
  DeletePictures 413
  DeleteStatePicture 416
  DeleteStatePictures 416
  EditLabel 432
  ExpandAll 439
  ExpandItem 439
  FindItem 473
  GetItem 543
  InsertItem 625, 626
  InsertItemFirst 627
  InsertItemLast 629
  InsertItemSort 631
  SelectItem 935
  SetDropHighlight 973
  SetFirstVisible 975
  SetItem 980
  SetLevelPictures 981
  SetOverlayPicture 988
  Sort 1042
  SortAll 1043
TrigEvent enumerated datatype 811
TriggerEvent function 1100
  triggering
    events 180
    functions or events 94
  TriggerPBEvent function 1102
  Trim function 1104
Truncate function 1105
TrustVerify function 1106
TypeOf function 1109
typographical conventions xxiv

U
Uncheck function 1110
Undo function 1112
Undo, testing 340
Unicode, string conversion 481, 1083, 1088
Uniform Data Transfer 505, 963
units
  converting from pixels 796
  converting to pixels 1113
  distance from edge 798
UnitsToPixels function 1113
unread messages 702
unsigned integer datatype 24
unsigned long datatype 24
UNTIL, in DO...LOOP statement 129
UPDATE statement 162
UPDATE Where Current of Cursor statement 164
UPDATEBLOB statement 163
Upper function 1115
UpperBound function 1115
uppercase 1115
user events
  defined 90
    pbm_dwngraphcreate 1004
user ID 705
user name 710
user objects
  about 79
    autoinstantiated 83
    closing 361
    closing tab page 360
    creating 125
    creating dynamically 126
    exporting as syntax 677
    listing 673
    opening 766, 767, 768, 774, 775, 776, 777
    pipeline 1046
    re-creating from syntax 678
Index

tab pages 766, 769
used like structures 83
user-defined events 179, 181
UserString event 301
displaying pop-up menus 800
setting 1034
visual user objects 79

V
value, passing arguments by 104
ValueChanged event 302
values
adding to lists 313
checking for NULL 654
data points 516
deleting from list 409
detecting numeric 655
inserting into lists 620
variables
access levels 41
assigning literals 20, 21, 22, 24
assigning values 38
checking for NULL 654
datatype 37
declaring 31
declaring initial values 38
default values 38
determining datatype of 349
extracting data from a blob 334
host 144
indicator 144
initializing with expression 39
inserting data into a blob 332
names 37
OLEObject 373
referencing in SQL 144
search order 33
setting to NULL 8, 987
validating 661
where to declare 32
variable-size arrays, memory allocation 50, 1115
vertical fill pattern 971, 1007
video monitor 527
ViewChange event 303
Visible property
and SetRedraw function 999

W
warm link 435, 560, 758, 1001
week, day of 401, 402
Which function 1117
WHILE, in DO...LOOP statement 129
white space 17
width
data point's line 969
series line 1005
setting 880
string 848
workspace 1121
Window ActiveX controls
GetArgElement function 486
GetLastReturn function 545
InvokePBFnction function 640
ResetArgElements function 877
SetArgElement function 952
TriggerPBEvent function 1102
Window functions
ArrangeSheets 325
ChangeMenu 343
ClassName 349
CloseUserObject 361
Draw 430
GetActiveSheet 484
GetFirstSheet 535
GetNextSheet 552
Hide 587
Move 734
Open 743
OpenSheet 760
OpenSheetWith Parm 762
OpenTab 766
OpenUserObject 774
OpenWith Parm 782
ParentWindow 790
PointerType 798
PointerType 799
PostEvent 811
Index

print 817
Resize 880
SetFocus 976
SetMicroHelp 986
SetPosition 993
SetRedraw 999
Show 1034
TriggerEvent 1100
TypeOf 1109
WorkSpaceHeight 1119
WorkSpaceWidth 1121
WorkSpaceX 1122
WorkSpaceY 1123

Window objects
  closing user objects 361
  exporting as syntax 677
  listing 673, 675
  recreating from syntax 678
Window painter 774, 776

windows
  adding user objects 766, 774, 777
  arranging 325, 760
  changing menus 343
  closing 356
  custom frames 1122, 1123
  datatype of 743
  DDE conversation handle 1054
  getting active 484
  obtaining handle 585
  obtaining workspace height 1119
  obtaining workspace width 1121
  opening 743, 782
  posting messages 810
  setting position of 993
WordCap function 1119
WordParm field
  and TriggerEvent function 1100
  posting events 811
workspace
  distance to screen 1122, 1123
  obtaining height of 1119
  obtaining width 1121
WorkSpaceHeight function 1119
WorkSpaceWidth function 1121
WorkSpaceX function 1122
WorkSpaceY function 1123

Write function 1124
Writes 1124

X
x value
  data point 503, 516, 732
  importing data 591, 594, 596, 598
  inserting from strings 599
XMLParseFile function 1125
XMLParseString function 1128
xValue enumerated datatype 503, 516

Y
y value
  data point 503, 516, 732
  importing data 591, 594, 596, 598
  inserting from strings 599
Year function 1131
year, about 398
Yield function 1132
yValue enumerated datatype 503, 516

Z
zero, determining 1039