Open Tools from Sybase, Inc.

PowerBuilder

PowerScript Reference: Volume 1

Version 6

Power Builder®

AA0519

October 1997

Copyright © 1991-1997 Sybase, Inc. and its subsidiaries.
All rights reserved.
Printed in Ireland.

Information in this manual may change without notice and does not represent a commitment on the part of Sybase, Inc. and its subsidiaries.

The software described in this manual is provided by Powersoft Corporation under a Powersoft License agreement. The software may be used only in accordance with the terms of the agreement.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc. and its subsidiaries.

Sybase, Inc. and its subsidiaries claim copyright in this program and documentation as an unpublished work, revisions of which were first licensed on the date indicated in the foregoing notice. Claim of copyright does not imply waiver of other rights of Sybase, Inc. and its subsidiaries.

ClearConnect, Column Design, ComponentPack, InfoMaker, ObjectCycle, PowerBuilder, PowerDesigner, Powersoft, S-Designor, SQL SMART, and Sybase are registered trademarks of Sybase, Inc. and its subsidiaries. Adaptive Component Architecture, Adaptive Server Anywhere, Adaptive Server Enterprise, Adaptive Warehouse, AppModeler, DataArchitect, DataExpress, Data Pipeline, DataWindow, dbQueue, ImpactNow, InstaHelp, Jaguar CTS, jConnect for JDBC, MetaWorks, NetImpact, Optima++, Power++, PowerAMC, PowerBuilder Foundation Class Library, Power J, PowerScript, PowerSite, Powersoft Portfolio, Powersoft Professional, PowerTips, ProcessAnalyst, Runtime Kit for Unicode, SQL Anywhere, The Model For Client/Server Solutions, The Future Is Wide Open, Translation Toolkit, UNIBOM, Unilib, Uninull, Unisep, Unistring, Viewer, WarehouseArchitect, Watcom, Watcom SQL Server, Web.PB, and Web.SQL are trademarks of Sybase, Inc. or its subsidiaries. Certified PowerBuilder Developer and CPD are service marks of Sybase, Inc. or its subsidiaries. DataWindow is a patented proprietary technology of Sybase, Inc. or its subsidiaries.

AccuFonts is a trademark of AccuWare Business Solutions Ltd.

All other trademarks are the property of their respective owners.

Contents

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

VOLUME 1

PART 1 **POWERSCRIPT TOPICS**

1 Language Basics **3**
- Comments 4
- Identifier names 6
- Labels 8
- Special ASCII characters 9
- NULL values 11
- Reserved words 13
- Pronouns 14
  - Parent 15
  - This 16
  - Super 17
- Statement continuation 19
- Statement separation 21
- White space 22

2 Data Types **23**
- Standard data types 24
- Any data type 29
- System object data types 32
- Enumerated data types 34

3 Declarations **35**
- Declaring variables 36
  - Where to declare variables 36
  - About using variables 37

Declaration syntax 40
Declaring constants 50
Declaring arrays 51
Values for array elements 54
Size of variable-size arrays 55
More about arrays 56
Declaring external functions 61
Data types for external function arguments 65
Calling external functions 69
Defining source for external functions 69
Declaring DBMS stored procedures as remote procedure calls (RPCs) 70

4 **Operators and Expressions 73**
Operators 74
Arithmetic operators 74
Relational operators 76
Concatenation operator 78
Operator precedence in expressions 79
Data type of expressions 80
Numeric data types 80
String and char data types 82

5 **Structures and Objects 85**
About structures 86
About objects 88
About user objects 88
Instantiating objects 90
Using ancestors and descendants 91
Managing memory 91
User objects that behave like structures 92
Assignment for objects and structures 93

6 **Calling Functions and Events 97**
About functions and events 98
Finding and executing functions and events 100
Triggering versus posting functions and events 102
Static versus dynamic calls 103
Overloading, overriding, and extending functions and events 109
Passing arguments to functions and events 112
Using return values of functions and events 114
Using cascaded calling and return values 115

Syntax for calling functions and events 117
Calling functions and events in an object's ancestor 121

PART 2 STATEMENTS, EVENTS, AND FUNCTIONS

7 **PowerScript Statements 127**
Assignment 128
CALL 131
CHOOSE CASE 132
CONTINUE 134
CREATE 135
DESTROY 139
DO...LOOP 140
EXIT 143
FOR...NEXT 144
GOTO 146
HALT 147
IF...THEN 148
RETURN 151

8 **SQL Statements 153**
Using SQL in scripts 155
CLOSE Cursor 158
CLOSE Procedure 159
COMMIT 160
CONNECT 161
DECLARE Cursor 162
DECLARE Procedure 163
DELETE 165
DELETE Where Current of Cursor 166
DISCONNECT 167
EXECUTE 168
FETCH 169
INSERT 170
OPEN Cursor 171
ROLLBACK 172
SELECT 173
SELECTBLOB 174
UPDATE 175
UPDATEBLOB 176
UPDATE Where Current of Cursor 178
Using dynamic SQL 179

Dynamic SQL Format 1 183
Dynamic SQL Format 2 184
Dynamic SQL Format 3 186
Dynamic SQL Format 4 189

9 PowerScript Events 195
About events 196
Activate 200
BeginDrag 201
BeginLabelEdit 205
BeginRightDrag 207
ButtonClicked 210
ButtonClicking 211
Clicked 212
Close 220
CloseQuery 222
ColumnClick 224
ConnectionBegin 226
ConnectionEnd 228
Constructor 229
DataChange 231
DBError 232
Deactivate 234
DeleteAllItems 235
DeleteItem 236
Destructor 238
DoubleClicked 239
DragDrop 244
DragEnter 250
DragLeave 252
DragWithin 254
EditChanged 258
EndLabelEdit 259
Error 261
ExternalException 264
FileExists 267
GetFocus 268
Hide 270
HotLinkAlarm 271
Idle 272
InputFieldSelected 273
InsertItem 274
ItemChanged 275
ItemChanging 278

ItemCollapsed 279
ItemCollapsing 280
ItemError 281
ItemExpanded 284
ItemExpanding 285
ItemFocusChanged 286
ItemPopulate 288
Key 289
LineDown 291
LineLeft 292
LineRight 293
LineUp 294
LoseFocus 295
Modified 297
MouseDown 299
MouseMove 302
MouseUp 306
Moved 309
Open 310
Other 313
PageDown 314
PageLeft 316
PageRight 317
PageUp 318
PictureSelected 320
PipeEnd 321
PipeMeter 322
PipeStart 323
PrintEnd 324
PrintFooter 325
PrintHeader 327
PrintPage 329
PrintStart 330
PropertyChanged 331
PropertyRequestEdit 332
RButtonDown 333
RButtonUp 336
RemoteExec 337
RemoteHotLinkStart 338
RemoteHotLinkStop 339
RemoteRequest 340
RemoteSend 341
Rename 342
Resize 343

RetrieveEnd 344
RetrieveRow 345
RetrieveStart 346
RightClicked 348
RightDoubleClicked 350
RowFocusChanged 352
RowFocusChanging 353
Save 355
ScrollHorizontal 356
ScrollVertical 358
Selected 359
SelectionChanged 360
SelectionChanging 363
Show 365
Sort 366
SQLPreview 369
SystemError 372
SystemKey 373
Timer 375
ToolbarMoved 377
UpdateEnd 378
UpdateStart 379
ViewChange 380

VOLUME 2

10 **PowerScript Functions 381**
Abs 382
AcceptText 383
Activate 385
AddCategory 387
AddColumn 389
AddData 391
AddItem 394
AddLargePicture 400
AddPicture 401
AddSeries 403
AddSmallPicture 405
AddStatePicture 406
Arrange 407
ArrangeSheets 408
Asc 410

Beep ... 412
Blob ... 413
BlobEdit ... 414
BlobMid ... 415
BuildModel ... 417
Cancel ... 420
CanUndo ... 421
CategoryCount ... 422
CategoryName ... 423
Ceiling ... 424
ChangeMenu ... 425
Char ... 426
Check ... 427
ClassList ... 428
ClassName ... 429
Clear ... 432
ClearValues ... 434
Clipboard ... 435
Close ... 438
CloseChannel ... 442
CloseTab ... 444
CloseUserObject ... 446
CloseWithReturn ... 448
CollapseItem ... 451
CommandParm ... 452
ConnectToNewObject ... 454
ConnectToNewRemoteObject ... 456
ConnectToObject ... 458
ConnectToRemoteObject ... 461
ConnectToServer ... 464
Copy ... 465
CopyRTF ... 467
Cos ... 469
Cpu ... 470
Create ... 471
CreateInstance ... 474
CreatePage ... 476
CrosstabDialog ... 477
Cut ... 478
DataCount ... 480
DataSource ... 481
Date ... 483
DateTime ... 487
Day ... 489

DayName 490
DayNumber 491
DaysAfter 492
DBCancel 494
DBErrorCode 497
DBErrorMessage 499
DBHandle 501
DebugBreak 502
Dec 503
DeleteCategory 504
DeleteColumn 505
DeleteColumns 506
DeleteData 507
DeletedCount 508
DeleteItem 510
DeleteItems 513
DeleteLargePicture 514
DeleteLargePictures 515
DeletePicture 516
DeletePictures 517
DeleteRow 518
DeleteSeries 519
DeleteSmallPicture 520
DeleteSmallPictures 521
DeleteStatePicture 522
DeleteStatePictures 523
Describe 524
DestroyModel 530
DirList 531
DirSelect 533
Disable 535
DisconnectObject 536
DisconnectServer 537
DoScript 538
Double 540
DoVerb 541
Drag 543
DraggedObject 545
Draw 546
EditLabel 548
Enable 550
EntryList 551
EventParmDouble 553
EventParmString 554

ExecRemote 555
Exp 559
ExpandAll 560
ExpandItem 561
Fact 562
FileClose 563
FileDelete 564
FileExists 565
FileLength 566
FileOpen 568
FileRead 571
FileSeek 574
FileWrite 575
Fill 577
Filter 578
FilteredCount 580
Find 582
FindCategory 588
FindClassDefinition 590
FindFunctionDefinition 592
FindGroupChange 593
FindItem 595
FindMatchingFunction 602
FindNext 605
FindRequired 606
FindSeries 610
FindTypeDefinition 612
GarbageCollect 614
GarbageCollectGetTimeLimit 615
GarbageCollectSetTimeLimit 616
GenerateHTMLForm 617
GetActiveSheet 619
GetAlignment 620
GetApplication 621
GetArgElement 622
GetAutomationNativePointer 623
GetBandAtPointer 625
GetBorderStyle 627
GetChanges 628
GetChild 631
GetChildrenList 634
GetClickedColumn 636
GetClickedRow 637
GetColumn 638

GetColumnName 641
GetCommandDDE 642
GetCommandDDEOrigin 644
GetCompanyName 645
GetContextKeywords 646
GetContextService 648
GetData 650
GetDataDDE 655
GetDataDDEOrigin 656
GetDataPieExplode 658
GetDataStyle 660
GetDataValue 667
GetDynamicDate 669
GetDynamicDateTime 671
GetDynamicNumber 673
GetDynamicString 674
GetDynamicTime 675
GetEnvironment 676
GetFileOpenName 677
GetFileSaveName 679
GetFirstSheet 681
GetFixesVersion 682
GetFocus 684
GetFormat 685
GetFullState 686
GetHostObject 688
GetItem 690
GetItemDate 694
GetItemDateTime 697
GetItemDecimal 699
GetItemNumber 702
GetItemStatus 705
GetItemString 707
GetItemTime 709
GetLastReturn 712
GetMajorVersion 714
GetMessageText 716
GetMinorVersion 717
GetName 719
GetNativePointer 720
GetNextModified 721
GetNextSheet 723
GetObjectAtPointer 725
GetOrigin 726

GetParagraphSetting 727
GetParent 728
GetRemote 730
GetRow 734
GetRowFromRowId 735
GetRowIdFromRow 737
GetSelectedRow 739
GetSeriesStyle 740
GetServerInfo 747
GetShortName 749
GetSpacing 750
GetSQLPreview 751
GetSQLSelect 752
GetStateStatus 754
GetText 756
GetTextColor 757
GetTextStyle 758
GetToolbar 759
GetToolbarPos 761
GetTrans 764
GetUpdateStatus 766
GetURL 769
GetValidate 770
GetValue 771
GetVersionName 773
GroupCalc 774
Handle 775
Hide 778
Hour 780
HyperLinkToURL 781
Idle 782
ImportClipboard 784
ImportFile 788
ImportString 793
IncomingCallList 798
InputFieldChangeData 800
InputFieldCurrentName 802
InputFieldDeleteCurrent 803
InputFieldGetData 804
InputFieldInsert 805
InputFieldLocate 806
InsertCategory 808
InsertClass 810
InsertColumn 811

InsertData 812
InsertDocument 815
InsertFile 817
InsertItem 818
InsertItemFirst 825
InsertItemLast 828
InsertItemSort 831
InsertObject 834
InsertPicture 835
InsertRow 836
InsertSeries 837
Int 839
Integer 840
InternetData 842
IntHigh 843
IntLow 844
InvokePBFunction 845
IsAllArabic 847
IsAllHebrew 848
IsAnyArabic 849
IsAnyHebrew 850
IsArabic 851
IsArabicAndNumbers 852
IsDate 853
IsHebrew 854
IsHebrewAndNumbers 855
IsNull 856
IsNumber 857
IsPreview 858
IsSelected 859
IsTime 861
IsValid 862
KeyDown 863
Left 867
LeftTrim 868
Len 869
Length 871
LibraryCreate 873
LibraryDelete 875
LibraryDirectory 877
LibraryExport 879
LibraryImport 881
LineCount 883
LineLength 885

LineList 886
LinkTo 887
Listen 889
Log 890
LogTen 891
Long 892
Lower 894
LowerBound 895
mailAddress 896
mailDeleteMessage 898
mailGetMessages 900
mailHandle 902
mailLogoff 903
mailLogon 904
mailReadMessage 906
mailRecipientDetails 909
mailResolveRecipient 911
mailSaveMessage 914
mailSend 917
Match 919
Max 923
MemberDelete 924
MemberExists 926
MemberRename 928
MessageBox 930
Mid 933
Min 936
Minute 937
Mod 938
ModifiedCount 939
Modify 941
ModifyData 955
Month 958
Move 959
MoveTab 961
NextActivity 962
Now 964
ObjectAtPointer 965
OLEActivate 968
Open 970
OpenChannel 987
OpenSheet 990
OpenSheetWithParm 993
OpenTab 996

OpenTabWithParm 1000
OpenUserObject 1005
OpenUserObjectWithParm 1009
OpenWithParm 1014
OutgoingCallList 1019
PageCount 1021
PageCreated 1022
ParentWindow 1023
Paste 1025
PasteLink 1027
PasteRTF 1029
PasteSpecial 1030
Pi 1031
PixelsToUnits 1032
PointerX 1033
PointerY 1034
PopMenu 1035
PopulateError 1037
Pos 1039
Position 1041
Post 1047
PostEvent 1049
PostURL 1053
Preview 1055
Print 1057
PrintBitmap 1066
PrintCancel 1068
PrintClose 1071
PrintDataWindow 1072
PrintDefineFont 1073
PrintLine 1075
PrintOpen 1077
PrintOval 1079
PrintPage 1081
PrintRect 1082
PrintRoundRect 1084
PrintScreen 1086
PrintSend 1087
PrintSetFont 1089
PrintSetSpacing 1090
PrintSetup 1091
PrintText 1092
PrintWidth 1094
PrintX 1095

PrintY 1096
ProfileInt 1097
ProfileString 1099
Rand 1101
Randomize 1102
Read 1103
Real 1106
RegistryDelete 1108
RegistryGet 1109
RegistryKeys 1111
RegistrySet 1113
RegistryValues 1116
RelativeDate 1117
RelativeTime 1118
ReleaseAutomationNativePointer 1119
ReleaseNativePointer 1120
RemoteStopConnection 1121
RemoteStopListening 1123
Repair 1124
Replace 1126
ReplaceText 1128
ReselectRow 1130
Reset 1131
ResetArgElements 1135
ResetDataColors 1137
ResetTransObject 1139
ResetUpdate 1140
Resize 1142
RespondRemote 1143
Restart 1145
Retrieve 1146
Reverse 1150
RGB 1151
Right 1153
RightTrim 1154
Round 1155
RoutineList 1156
RowCount 1157
RowsCopy 1159
RowsDiscard 1161
RowsMove 1162
Run 1164
Save 1166
SaveAs 1168

SaveAsAscii 1178
SaveDocument 1180
Scroll 1182
ScrollNextPage 1183
ScrollNextRow 1186
ScrollPriorPage 1189
ScrollPriorRow 1191
ScrollToRow 1194
Second 1197
SecondsAfter 1198
Seek 1199
SelectedColumn 1201
SelectedIndex 1202
SelectedItem 1203
SelectedLength 1204
SelectedLine 1206
SelectedPage 1208
SelectedStart 1209
SelectedText 1211
SelectItem 1213
SelectObject 1217
SelectRow 1218
SelectTab 1219
SelectText 1220
SelectTextAll 1225
SelectTextLine 1226
SelectTextWord 1227
Send 1229
SeriesCount 1231
SeriesName 1232
SetActionCode 1234
SetAlignment 1236
SetArgElement 1237
SetAutomationLocale 1239
SetAutomationPointer 1241
SetAutomationTimeout 1243
SetBorderStyle 1245
SetChanges 1246
SetColumn 1249
SetConnect 1252
SetData 1254
SetDataDDE 1256
SetDataPieExplode 1258
SetDataStyle 1260

SetDetailHeight 1267
SetDropHighlight 1269
SetDynamicParm 1270
SetFilter 1272
SetFirstVisible 1275
SetFocus 1276
SetFormat 1277
SetFullState 1278
SetItem 1280
SetItemStatus 1286
SetLevelPictures 1289
SetLibraryList 1291
SetMask 1293
SetMicroHelp 1295
SetNull 1296
SetOverlayPicture 1297
SetParagraphSetting 1299
SetPicture 1300
SetPointer 1301
SetPosition 1303
SetProfileString 1306
SetRedraw 1308
SetRemote 1310
SetRow 1313
SetRowFocusIndicator 1315
SetSeriesStyle 1317
SetSort 1325
SetSpacing 1327
SetSQLPreview 1328
SetSQLSelect 1329
SetState 1331
SetTabOrder 1332
SetText 1333
SetTextColor 1335
SetTextStyle 1336
SetToolbar 1338
SetToolbarPos 1340
SetTop 1345
SetTraceFileName 1346
SetTrans 1348
SetTransObject 1350
SetTransPool 1354
SetValidate 1356
SetValue 1358

ShareData 1360
ShareDataOff 1363
SharedObjectDirectory 1364
SharedObjectGet 1365
SharedObjectRegister 1367
SharedObjectUnregister 1368
Show 1369
ShowHeadFoot 1370
ShowHelp 1371
Sign 1373
SignalError 1374
Sin 1376
Sort 1377
SortAll 1381
Space 1383
Sqrt 1384
Start 1385
StartHotLink 1393
StartServerDDE 1395
State 1397
Stop 1399
StopHotLink 1400
StopListening 1402
StopServerDDE 1403
String 1404
SyntaxFromSQL 1410
SystemRoutine 1413
TabPostEvent 1414
TabTriggerEvent 1415
Tan 1416
Text 1417
TextLine 1418
Time 1419
Timer 1422
ToAnsi 1424
Today 1425
Top 1426
TotalColumns 1427
TotalItems 1428
TotalSelected 1429
ToUnicode 1430
TraceBegin 1431
TraceClose 1433
TraceDisableActivity 1434

TraceEnableActivity 1436
TraceEnd 1438
TraceError 1439
TraceOpen 1440
TraceUser 1443
TriggerEvent 1444
TriggerPBEvent 1446
Trim 1448
Truncate 1449
TypeOf 1450
Uncheck 1452
Undo 1454
UnitsToPixels 1455
Update 1456
UpdateLinksDialog 1460
Upper 1462
UpperBound 1463
WorkSpaceHeight 1466
WorkSpaceWidth 1468
WorkSpaceX 1469
WorkSpaceY 1470
Write 1471
Year 1473
Yield 1474

About This Book

Subject

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

Audience

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

PART 1

PowerScript Topics

CHAPTER 1 Language Basics

About this chapter

This chapter describes general elements and conventions of PowerScript.

Contents

Topic	Page
Comments	4
Identifier names	6
Labels	8
Special ASCII characters	9
NULL values	11
Reserved words	13
Pronouns	14
Statement continuation	19
Statement separation	21
White space	22

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* */

Usage

Here is how to use each method:

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

Adding comment markers
In the PowerScript and Function painters, 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 INFO 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

Description

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 13)
- Can have up to 40 characters but no spaces
- Are case insensitive (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. This means that when you use the subtraction operator or the -- operator in a script, you must surround it with spaces (otherwise, PowerBuilder thinks the expression is 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. This way 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
Actual-$-amount
```

```
Part#
```

Invalid identifiers

```
2nd-quantity // Does not start with a letter
ABC Code     // Contains a space
Child'sId    // Contains invalid special character
```

Labels

Description

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

Syntax

Identifier :

Usage

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

FOR INFO For information about the GOTO statement, see GOTO on page 146. For information about valid identifiers, see "Identifier names" on page 6.

Examples

On a line by itself above the statement

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

At the start of the line before the statement

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

Special ASCII characters

Description

You can include special ASCII characters in strings. For example, you may 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.

Syntax

Follow these guidelines:

In this category	To specify this	Enter this	More information
Common ASCII characters	Newline	~n	
	Tab	~t	
	Vertical tab	~v	
	Carriage return	~r	
	Formfeed	~f	
	Backspace	~b	
	Double quote	~"	
	Single quote	~'	
	Tilde	~~	
Any ASCII character	Decimal	~###	### = a 3-digit number from 000 to 255
	Hexadecimal	~h##	## = a 2-digit hexadecimal number from 01 to FF
	Octal	~o###	### = a 3-digit octal number from 000 to 377

Examples

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

String	Description
"dog~n"	A string containing the word dog followed by a newline character
"dog~tcat~ttiger"	A string containing the word *dog*, a tab character, the word *cat*, another tab character, and the word *tiger*

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:

Value	Description
"~249"	The ASCII character with decimal value 249
"~hF9"	The ASCII character with hexadecimal value F9
"~o371"	The ASCII character with octal value 371

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 data types, 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 data type—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 NOT TRUE.

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 will make the computer beep (the variable nbr is set to NULL, so each statement evaluates to NOT TRUE):

```
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 NOT TRUE, so the ELSE is executed:

```
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 NOT TRUE, and the ELSE is executed:

```
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 will get a compiler warning.

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

The PowerBuilder system class also includes private variables that cannot be used as identifiers. If you use a private variable as an identifier, you will 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.

Syntax

These are the pronouns:

- Parent
- This
- Super

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 individual pronouns Each pronoun has a specific meaning and use:

This pronoun	In a script for a	Refers to the
This	Window, custom user object, menu, application object, or control	Object or control itself
Parent	Control in a window	Window containing the control
	Control in a custom user object	Custom user object containing the control
	Menu	Item in the menu on the level above the current menu
Super	Descendent object or control	Parent
	Descendent window or user object	Immediate ancestor of the window or user object
	Control in a descendent window or user object	Immediate ancestor of the control's parent window or user object

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.

Parent

Description

Parent 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 scrollbar 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 checkbox hides the user object:

```
Parent.Hide( )
```

If you include this statement in the script for the CheckBox, clicking the checkbox 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

Description

The pronoun This 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. This 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.

Examples

Example 1 This statement in a script for a menu places a checkmark 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). But 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, the following statement in the script for the Clicked event for a CommandButton means that clicking the button changes the horizontal position of the button (changes the button's X property):

```
This.x = This.x + 50
```

Super

Description

When you write a 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 can't 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 will return a syntax error.

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

FOR INFO See the discussion of CALL on page 131.

Examples

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

```
Super::wf_myfunc(myarg1, myarg2)
```

This example has to be part of a script or function in the descendent window, not one of the window's controls. For example, if it is in the Clicked event of a button on the descendent 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:

```
Super::EVENT Clicked()
```

Statement continuation

Description

Although you typically put one statement on each line, you will occasionally want to continue a statement to more than one line. The statement continuation character is the ampersand (&).

Syntax

Start of statement &
  more statement &
  end of statement

The ampersand must be the last nonwhite character on the line (or the compiler will consider it part of the statement).

FOR INFO For information about white space, see "White space" on page 22.

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 You can continue a quoted string by simply placing an ampersand in the middle of the string and continuing the string on the next line:

```
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 This 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 may result in an error:

```
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 will cause an error.This statement will fail, 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 will 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, formfeeds, and comments are forms of white space. The compiler treats white space as a delimiter and doesn't 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 6), you must surround a dash used as a minus sign with spaces. Otherwise, PowerBuilder considers the dash 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:"
```

CHAPTER 2 Data Types

About this chapter

This chapter describes the PowerScript data types.

Contents

Topic	Page
Standard data types	24
Any data type	29
System object data types	32
Enumerated data types	34

Standard data types

The data types

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

These are the standard PowerScript data types:

Blob	Integer or Int
Boolean	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.

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 may want to use char arrays instead of strings.

FOR INFO For more information about passing character-based data to external functions, see *Application Techniques*. For information about data type conversion when assigning strings to chars and vice versa, see "String and char data types" on page 82.

Using literals Char is one of the standard data types for which you can use literals to assign values. 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 Date is one of the standard data types for which you can use literals to assign values. To assign a literal value, separate the year, month, and day with hyphens. For example:

```
1992-12-25  // December 25, 1992
1995-02-06  // February 6, 1995
```

DateTime

The date and time in a single data type, used only for reading and writing DateTime values from and to a database. To convert DateTime values to data types 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 18 digits. You can place the decimal point anywhere within the 18 digits—for example, 123.456, 0.000000000000000001 or 12345678901234.5678.

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

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

Double

A signed floating-point number with 15 digits of precision and a range from 2.2250738585072E-308 to 1.79769313486232E+308.

Integer or Int

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

Using literals Integer is one of the standard data types for which you can use literals to assign values. To assign a literal value, use any whole number (positive, negative, or zero). The leading plus sign is optional (18 and +18 are the same). For example:

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

Long

32-bit signed integers, from -2,147,483,648 to +2,147483,647.

Real

A signed floating-point number with six digits of precision and a range from 1.175494E-38 to 3.402823E+38.

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

```
2E4         2.5E78    +6.02E3    -4.1E-2
-7.45E16    7.7E+8    3.2E-45
```

String

Any ASCII character with variable length (0 to 2,147483,647).

16-bit PowerBuilder
In 16-bit PowerBuilder the length of a string is still limited to 60,000 characters.

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 INFO For more information about passing character-based data to external functions, see *Application Techniques*. For information about data type conversion when assigning strings to chars and vice versa, see "String and char data types" on page 82.

Using literals String is one of the standard data types for which you can use literals to assign values. 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:

```
string s1
s1 = "This is a string"
s1 = '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:

```
string s1
s1 = "Here's a string."
```

results in the string *Here's a string*.

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

```
string s1 = '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 figure out the quotation marks, consider how PowerBuilder will parse the string. The following string is a possible argument for the Modify function; it mixes single and double quotes to reduce the number of tildes:

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

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

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

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

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

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

Rules for quotation marks and tildes When nesting quoted strings, the following rules of thumb may 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.

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

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

UnsignedInteger, UnsignedInt, or UInt

16-bit unsigned integers, from 0 to 65,535.

UnsignedLong or ULong

32-bit unsigned integers, from 0 to 4,294,967,295.

Any data type

General information

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

Declarations and assignments

You declare Any variables just as you do any other variable.

You assign data to Any variables with standard assignment statements. You can assign an array to a simple Any variable. You can also declare an array of Any variables, where each element of the array can have a different data type.

After you assign a value to an Any variable, you can test the variable with the ClassName function and find out the actual data type:

```
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 data type.

Restrictions

If the value of a simple Any variable is an array, you can't access the elements of the array until you assign the value to an array variable of the appropriate data type. 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 can't use dot notation to access the elements of the structure until you assign the value to a structure of the appropriate data type.

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

Operations and expressions

You can perform operations on Any variables as long as the data type of the data in the Any variable is appropriate to the operator. If the data type isn't 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:

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

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

```
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.

Data type conversion functions PowerScript data type 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:

```
ls_string = ia_any
```

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

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

Other functions If a function's prototype does not allow Any as a data type for an argument, you cannot use an Any variable without a conversion function, even if it contains a value of the correct data type. When you compile the script, you will 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:

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

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

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

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

```
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 changes too.

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

Overusing the Any data type

Do not use Any variables as a substitute for selecting the correct data type 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 data types 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 data types are correct before code gets executed. With Any variables, errors that can be caught by the compiler are not found until the code is run.

System object data types

Objects as data types

System object data types are specific to PowerScript. You can list 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 data type. Usually you don't need to concern yourself with these objects as data types—you simply define the objects in a PowerBuilder painter and use them.

But sometime you need to understand how PowerBuilder maintains its system objects in a hierarchy of data types. For example, when you need to define instances of a window, you will define variables whose data type is window. When you need to create an instance of a menu to pop up in a window, you will define a variable whose data type 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 data types that you can use in your applications. You can define variables whose type is any class.

For example, to define a window variable, you could code:

```
window mywin
```

To define a menu variable, you could code:

```
menu mymenu
```

If you have a series of buttons in a window and for some reason need to keep track of one of them (such as the last one clicked), you could declare a variable of type CommandButton and assign it the appropriate button in the window:

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

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

FOR INFO To learn more about working with instances of objects through data types, see "About objects" on page 88.

Enumerated data types

About enumerated data types

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

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

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

A variable of one of the enumerated data types can be assigned a fixed set of values. Values of enumerated data types always end with an exclamation point (!).

For example, the enumerated data type Alignment, which specifies the alignment of text, can be assigned one of the following three values: Center!, Left!, and Right!:

```
mle_edit.Alignment=Right!
```

Incorrect syntax
Do not enclose an enumerated data type value in quotation marks or you will receive a compiler error.

Advantages of enumerated types

Enumerated data types have the following advantage over standard data types: when an enumerated data type is required, the compiler checks the data and makes sure it is the correct type.For example, if you set an enumerated data type variable to any other data type or to an incorrect value, the compiler will not allow it.

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 (**RPC**s) and external functions that reside in dynamic link libraries (**DLL**s).

Contents

Topic	Page
Declaring variables	36
Declaring constants	50
Declaring arrays	51
Declaring external functions	61
Declaring DBMS stored procedures as remote procedure calls (RPCs)	70

Declaring variables

General information

Before you use a variable in a script, you must declare it (give it a data type and a name).

A variable can be a standard data type, a structure, or an object. Object data types 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.

Where to declare variables

Scope

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

These are the four scopes of variables.

Scope	Description
Global	Accessible anywhere in the application. It is independent of any object definition.
Instance	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. Instance variables can belong to the application object, a window, a user object, or a menu.
Shared	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. Shared variables can belong to the application object, a window, a user object, or a menu.
Local	A temporary variable that is accessible only in the script in which you define it. When the script is finished, the variable constant ceases to exist.

Global declarations

Global variables can be defined in the Window, User Object, Menu, or PowerScript painter for any object.

Instance and shared declarations

Instance and shared variables belong to particular objects: windows, user objects, menus, or the application object. Before you can declare them, you must open the object in its painter:

- **Window objects** In the Window painter, open the window for which you want to declare a variable. Optionally, open a script for the window or a control in the window.
- **User objects** In the User Object painter, open the user object for which you want to declare a variable. Optionally, open a script for the user object or a control in the user object.
- **Menu objects** In the Menu painter, open the menu for which you want to declare a variable. Optionally, open a script for one of the menu items.
- **Application objects** In the Application painter, open a script for the application object.

Local declarations

You declare local variables in the script itself.

Declaring SQL cursors

You can also declare SQL cursors that are global, shared, instance, or local. Type the DECLARE SQL statement in the appropriate dialog or in the script.

About using variables

General information

To use or set a variable's value in a 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 may 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
- 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 such as:

```
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

(Of course, 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:

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

Naming errors If there is a local or global variable 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 will get an informational message from the compiler if you use the same name for instance and global variables.

Declaration syntax

Simple syntax

In its simplest form, a variable declaration requires only two parts: the data type and the variable name:

datatype *variablename*

Full syntax

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

{ *access* } *datatype* { *{ size }* } { *{ precision }* } variablename { = *value* }
{, *variablename2* { = *value2* } }

Parameter	Description
access (optional)	(For instance variables only) Keywords specifying the access for the variable FOR INFO For information, see "Access for instance variables" on page 45
datatype	The data type of the variable. You can specify a standard data type, 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
{size} (optional)	(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 during execution If you enter a size that exceeds the declared length in a script, PowerBuilder truncates the blob data
{precision} (optional)	(For decimals only) A number, enclosed in braces, specifying the number of digits after the decimal point; if you don't 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 6) (For arrays only) You can define additional variables with the same data type by naming additional variable names, separated by commas; each variable can have a value

Parameter	Description
value (optional)	A literal or expression of the appropriate data type that will be the initial value of the variable Blobs cannot be initialized with a value FOR INFO For information, see "Initial values" on page 42

Examples

Declaring instance variables

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

Declaring a global variable

```
string gs_name
```

Declaring shared variables

```
time st_process_start
string ss_process_name
```

Declaring local variables

```
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:

```
blob ib_Emp_Picture
```

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

```
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:

```
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 doesn't 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
```

Data type

A variable can be declared as one of the following data types:

- A standard data type (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

General information

In a well-planned application, standards determine how you will name your 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 data type 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 INFO For information about naming conventions, see *Application Techniques*.

X and Y as variable names

Although you may 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 will *not* get a compiler error. PowerBuilder will assume you want to move the object, which may lead to unexpected results in your application.

Initial values

General information

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

Default values for variables

If you do not initialize a variable when you declare it, PowerBuilder sets the variable to the default value for its data type as follows:

For this variable data type	PowerBuilder sets this default value
Blob	A blob of 0 length; an empty blob
Char (or character)	ASCII value 0
Boolean	FALSE
Date	1900-01-01 (January 1, 1900)
DateTime	1900-01-01 00:00:00
Numeric (integer, long, decimal, real, double, UnsignedInteger, and UnsignedLong)	0
String	Empty string ("")
Time	00:00:00 (midnight)

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 data type after the variable. For information about literals for specific data types, see "Standard data types" on page 24.

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

```
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:

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

This example initializes ls_method with the string "UPS":

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

This example initializes ls_headers to three words separated by tabs:

```
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:

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

This example declares ld_StartDate as a date and initializes it with the date Feb 1, 1993:

```
date ld_StartDate = 1993-02-01
```

Specifying an expression as an initial value

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

```
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 during execution.

If the expression's value changes Because the expression's value is set to the variable when the script is compiled (not during execution) 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.

Unwanted result In this declaration, the value of d is the date *the script is compiled*:

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

Wanted result In contrast, these statements result in d being set to the date *the application is run*:

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

How shared variables are initialized

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

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

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

How instance variables are initialized

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

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

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

When to use shared variables instead Use a shared variable instead of an instance variable if you need a variable that:

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

Access for instance variables

Description

The general syntax for declaring variables (see "Declaration syntax" on page 40) 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*

Parameter	Description
access-right (optional)	A keyword specifying where the variable's name will be recognized. Values are: ♦ 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 ♦ PROTECTED — Scripts for the object for which the variable is declared and its descendants can refer to the variable ♦ 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
readaccess (optional)	A keyword restricting the ability of scripts to read the variable's value. Values are: ♦ PROTECTEDREAD — Only scripts for the object and its descendants can read the variable ♦ PRIVATEREAD — Only scripts for the object can read the variable 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
writeaccess (optional)	A keyword restricting the ability of scripts to change the variable's value. Values are: ♦ PROTECTEDWRITE — Only scripts for the object and its descendants can change the variable ♦ PRIVATEWRITE — Only scripts for the object can change the variable When *access-right* is PUBLIC, you can specify either keyword. When *access-right* is PROTECTED, you can specify only PRIVATEWRITE. You cannot specify a modifier for PRIVATE access, because PRIVATE is already fully restricted If *writeaccess* is omitted, any script can change the variable
datatype	A valid data type. See "Declaration syntax" on page 40
variablename	A valid identifier. See "Declaration syntax" on page 40

Usage

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

```
public protectedwrite integer ii_count
```

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

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

Examples

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

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

```
private integer ii_a, ii_n
public  integer ii_Subtotal
protected integer ii_WinCount
```

This protected variable can only be changed by scripts of the owner object; descendants of the owner can read it:

```
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 privateread privatewrite 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):

```
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:

```
public privatewrite integer ii_int
```

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

```
w_emp w_myemp
w_myemp.ii_int = 1 // Cannot write to variable
```

Another format for access-right keywords

Description

You can also group declarations 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 declaration of a standard data type 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*

Parameter	Description
CONSTANT	Declares a constant instead of a variable. The CONSTANT keyword can be before or after the *access* keywords
access (optional)	(For instance variables only) Keywords specifying the access for the constant FOR INFO For information, see "Access for instance variables" on page 45
datatype	A standard data type 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 INFO For information about PowerBuilder data types, see "Standard data types" on page 24
constname	The name of the constant (must be a valid PowerScript identifier, as described in "Identifier names" on page 6)
value	A literal or expression of the appropriate data type that will be the value of the constant. The value is required FOR INFO For information, see "Initial values" on page 42

Usage

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

Examples

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

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

Declaring arrays

Description

An array is an indexed collection of elements of a single data type. 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 approximately two gigabytes of elements (2,147,483,647 to be exact).

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

Syntax

{ *access* } *datatype variablename [* { *d1, ..., dn* } *]* { = *{ valuelist }* }

Parameter	Description
access (optional)	(For instance variables only) Keywords specifying the access for the variable FOR INFO For information, see "Access for instance variables" on page 45
datatype	The data type of the variable. You can specify a standard data type, 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 "Declaration syntax" on page 40): decimal {2} *variablename*[ ] For blobs, fixed-length blobs within an array are not supported. If you specify a size after *datatype*, it is ignored
variablename	The name of the variable (name must be a valid PowerScript identifier, as described in "Identifier names" on page 6) You can define additional arrays with the same data type by naming additional variable names with brackets and optional value lists, separated by commas

Parameter	Description
[{ *d1*, ..., *dn* }]	Brackets and (for fixed-size arrays) one or more integer values (*d1* through *dn*, one for each dimension) specifying the sizes of the dimensions For a variable-size array, which is always 1-dimensional, specify brackets only FOR INFO For more information on how variable-size arrays change size, see "Size of variable-size arrays" on page 55 For a fixed-size array, the number of dimensions is determined by the number of integers you specify and is limited only by the amount of available memory For fixed-size arrays, you can use TO to specify a range of element numbers (instead of a dimension size) for one or more of the dimensions. Specifying TO allows you to change the lower bound of the dimension (*upperbound* must be greater than *lowbound*): *[d1lowbound* TO *d1upperbound* {, ... , *dnlowbound* TO *dnupperbound* } *]*
{ *valuelist* } (optional)	A list of initial values for each position of the array. The values are separated by commas and the whole list is enclosed in braces. The number of values cannot be greater than the number of positions in the array. The data type of the values must match *datatype*

Examples

These declarations create variable-size arrays:

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

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

```
dec lc_limit[ ]
```

Fixed arrays These declarations create fixed-size, 1-dimensional arrays:

```
integer li_TaxCode[3]     // Array of 3 integers.
string ls_day[7]          // Array of 7 strings.
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:
                             // Rate[2] through Rate[5]
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 6-element, 2-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:

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

More declarations of multidimensional arrays:

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

This declaration creates three decimal arrays:

```
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 data type. For example, in a newly declared integer array:

```
integer li_TaxCode[3]
```

the elements li_TaxCode[1], li_TaxCode[2], and li_TaxCode[3] are all initialized to zero.

FOR INFO For information about default values for basic data types, see "Initial values" on page 42.

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 don't have to initialize all the elements of the array, but you can't 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.

Assigning values

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

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

Examples

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

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

Example 2 This statement initializes a 2-dimensional array:

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

As a result:

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

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

```
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 1-dimensional arrays can be variable-size arrays.

Because you don't declare the size, you can't use the TO notation to change the lower bound of the array. So the lower bound of a variable-size array is always 1.

How memory is allocated

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

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

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

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

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

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

More about arrays

This section provides technical details about:

- Assigning one array to another
- Using 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

Assigning to an unbounded array The target array is the same as the source:

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

Assigning 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:

```
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:

```
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 1-dimensional array. PowerBuilder then uses the rules for 1-dimensional arrays (described above) to assign the array to the target.

Not all array assignments are allowed, as described in the following rules.

Assigning one multidimensional array to another If the dimensions of the two arrays match, the target array becomes an exact copy of the source:

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

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

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

Assigning a 1-dimensional array to a multidimensional array A 1-dimensional array can be assigned to a multidimensional array. The values are mapped onto the multidimensional array in column major order:

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

```
b = a
```

Multidimensional to 1-dimensional array A multidimensional array can also be assigned to a 1-dimensional array. The source is linearized in column major order and assigned to the target:

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

Examples

Suppose you declare three arrays (a, b, and c). One(c) is unbounded and 1-dimensional; the other two (a and b) are multidimensional with different dimensions:

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

Array b is laid out like this:

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]

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

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

This statement explicitly linearizes b into c:

```
c = b
```

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

```
a = c
```

The values in array a are laid out like this:

1 for a[1,1]	3 for a[1,2]
2 for a[2,1]	4 for a[2,2]

Initializing a with an arraylist produces the same result:

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

Using arraylists to assign values to an array

General information

An arraylist is a list of values enclosed in braces used to initialize arrays. An arraylist represents a 1-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 56.

Examples

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

```
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):

```
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):

```
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:

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

1	4
2	5
3	6

If you think of a 3-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}
```

1	3		5	0
2	4		6	0

Errors that occur when addressing arrays

Fixed-size arrays

Referring to array elements outside the declared size causes an error during execution—for example:

```
int test[10]
test[11]=50         // This causes an execution error.
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—that is how the array works. However, accessing a variable-size array above its largest assigned value or below its lower bound causes an error during execution:

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

Declaring external functions

Description

External functions are functions that are written in languages other than PowerScript and stored in dynamic link libraries (DLLs), known as shared libraries on Macintosh and UNIX. 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.

Two types You can declare two types of external functions:

- **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.

FOR INFO 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:

{ *access* } FUNCTION *returndatatype name* ({ { REF } *datatype1 arg1*, ..., { REF } *datatypen argn* }) LIBRARY *"libname"* ALIAS FOR *"extname"*

External subroutine syntax You can also declare external subroutines (which are the same as external functions except that they don't return a value) using this syntax:

{ *access* } SUBROUTINE *name* ({ { REF } *datatype1 arg1*, ..., { REF } datatypen argn}) LIBRARY *"libname"* ALIAS FOR *"extname"*

Parameter	Description
access (optional)	(Local external functions only) You can specify Public, Protected, or Private to specify the access level of a local external function (the default is Public) FOR INFO For more information, see the section about specifying access of local functions in "Usage" next
FUNCTION or SUBROUTINE	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

Parameter	Description
returndatatype	The data type of the value returned by the function
name	The name of a function or subroutine that resides in a DLL
REF	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
datatype arg	The data type 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 INFO For more information on passing arguments, see *Application Techniques*
LIBRARY *"libname"*	The LIBRARY keyword is followed by a string containing the name of the DLL in which the function or subroutine is stored. Microsoft Windows DLLs usually have the extension DLL or EXE **On Windows** *libname* is a dynamic link library, which is a file that usually has an extension DLL or EXE **On Macintosh** *libname* is the name of a fragment defined when you built the shared library, not the name of the file **On UNIX** *libname* is a shared library, which is a file that usually has the extension .so **Naming** Instead of following different naming conventions on each platform, you could use the same library name on all platforms so that function declarations are the same. Enclose the library name in quotation marks and do not include the path. The library must be available to the application at execution time

Parameter	Description
ALIAS FOR *"extname"* (optional)	Keywords followed by a string giving the name of the function as defined in the DLL. If the name in the DLL 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 *"extname"* to establish the association between the PowerScript name and the external name

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:

Access level	Where you can use the local external function
Public	Any script in the application
Private	Scripts for events in the object for which the function is declared. You cannot use the function in descendants of the object
Protected	Scripts for the object for which the function is declared and its descendants

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

Availability of the dynamic library during execution To be available to the 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

On Macintosh The shared library is usually stored in the Extensions folder. For more information, see the section on other processing extensions in *Application Techniques*.

On UNIX Update the user's LD_LIBRARY_PATH environment variable to list the directory where the shared library is stored.

Examples

In the examples50 application that comes with PowerBuilder, these 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 since they are part of the same user object, you don't 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 WIN32 DLL WINMM.DLL:

```
//playsound
FUNCTION boolean sndPlaySoundA (string SoundName,
uint Flags) LIBRARY "WINMM.DLL"
FUNCTION uint waveOutGetNumDevs () LIBRARY "WINMM.DLL"
```

The declarations for WIN16 versions of the functions are similar:

```
//playsound
FUNCTION boolean sndPlaySound (string SoundName, uint
Flags) LIBRARY "mmsystem.dll"
FUNCTION uint waveOutGetNumDevs () LIBRARY
"mmsystem.dll"
```

A function called uf_playsound in the examples50 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 Win32 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
//don't 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
//don't 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 Win32 GetSysColor function:

```
FUNCTION ulong GetSysColor (int index) LIBRARY
"USER32.DLL"
```

This statement calls the external function. The meanings of the index argument and the return value are specified in the Win32 documentation:

```
RETURN GetSysColor (ai_index)
```

Example 3 This is the declaration for the Win32 GetSysColor function:

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

This statement calls the external function to get the screen height:

```
RETURN GetSystemMetrics(1)
```

This statement calls the external function to get the screen width:

```
RETURN GetSystemMetrics(0)
```

Data types for external function arguments

General information

When you declare an external function, the data types of the arguments must correspond with the data types as declared in the function's source definition.

What's here

This section documents the correspondence between data types in external functions and data types in PowerBuilder. It also includes information on byte alignment when passing structures by value.

What data type to use

How to use this section

Use the tables in this section to find out what PowerBuilder data type to use in an external function declaration. The PowerBuilder data type you select depends on the data type in the source code for the function and sometimes on whether you are running on a 16-bit or 32-bit platform.

Here's how the tables work:

- **Columns** The first column lists data types in source code. The second column describes the data type so you know exactly what it is. The third column lists the PowerBuilder data type you should use in the external function declaration.
- **Platform information** When the platform affects your choice, the platform is included in the first column. Platform notes are provided where appropriate:
- The 32-bit platforms that PowerBuilder supports are Windows 95, Windows NT, Macintosh, and UNIX (Solaris)
- The 16-bit platform that PowerBuilder supports is Windows 3.1

Pointers

Data type in source code	Size, sign, precision	PowerBuilder data type
* (any pointer)	32-bit pointer	Long
char *	Array of bytes of variable length	Blob

Platform notes

Windows pointers 32-bit FAR pointers such as LPBYTE, LPDWORD, LPINT, LPLONG, LPVOID, and LPWORD are declared in PowerBuilder as long.

On Windows 3.1 (a 16-bit platform), HANDLE is defined as 16 bits unsigned and is declared in PowerBuilder as an UnsignedInteger.

On Windows NT and other 32-bit Windows platforms, HANDLE is defined as 32 bits unsigned and is declared in PowerBuilder as an UnsignedLong.

Near-pointer data types on Windows (such as PSTR and NPSTR) are not supported in PowerBuilder.

Macintosh pointers The 32-bit pointers Ptr and Handle are declared in PowerBuilder as UnsignedLong.

Characters and strings

Data type in source code	Size, sign, precision	PowerBuilder data type
char	8 bits, signed	Char
string	32-bit pointer to a null-terminated array of bytes of variable length	String

Platform notes

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

Boolean

Platform notes

BOOL in Windows BOOL on Windows is 16-bit, signed. It is declared in PowerBuilder as boolean.

Boolean for Macintosh Boolean on Macintosh is 8-bit, unsigned. It is declared in PowerBuilder as char.

Fixed-point values

Data type in source code	Size, sign, precision	PowerBuilder data type
short	16 bits, signed	Integer
unsigned short	16 bits, unsigned	UnsignedInteger

Data type in source code	Size, sign, precision	PowerBuilder data type
int (16-bit platforms)	16 bits, signed	Integer
int (32-bit platforms)	32 bits, signed	Long
unsigned int (16-bit platforms)	16 bits, unsigned	UnsignedInteger
unsigned int (32-bit platforms)	32 bits, unsigned	UnsignedLong
long	32 bits, signed	Long
unsigned long	32 bits, unsigned	UnsignedLong

Platform notes

WORD and DWORD on Windows The Windows definition WORD is declared in PowerBuilder as UnsignedInteger and the Windows definition DWORD is declared as an UnsignedLong.

Short on 32-bit platforms On 32-bit platforms, you can't call external functions with return values or arguments of type short.

Floating-point values

Data type in source code	Size, sign, precision	PowerBuilder data type
float	32 bits, single precision	Real
double	64 bits, double precision	Double

Platform notes

Doubles on Windows PowerBuilder does not support 80-bit doubles on Windows.

Doubles on Macintosh PowerBuilder does not support extended doubles on Macintosh.

Notes for data types

Str255

On the Macintosh, the Pascal string data type Str255 is not supported in PowerBuilder. Strings are passed to external functions as C strings.

Date and time

The PowerBuilder data types 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 as long as they have the same definitions and alignment as the structure's components.

The DLL or shared library must be compiled using byte alignment, meaning no padding is added to align fields within the structure.

Calling external functions

Global external functions

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

You 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 functions and events" on page 117.

Defining source for external functions

General information

When you create your own functions for use as external functions in PowerBuilder external function calls, you must create the functions using the FAR PASCAL declaration and link them in a DLL.

For information

For more information about using external functions, see *Application Techniques*.

Declaring DBMS stored procedures as remote procedure calls (RPCs)

Description

You can use dot notation for calling non-result-set stored procedures:

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

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*" }

Argument	Description
FUNCTION or SUBROUTINE	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
rtndatatype	In a FUNCTION declaration, the data type of the value returned by the function
functionname	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
REF	Specifies that you are passing by reference the argument that follows REF. The stored procedure can store a value in *arg* that will be accessible to the rest of the PowerBuilder script
datatype arg	The data type and name of the arguments for the stored procedure. The list must match the definition of the stored procedure in the database. Each *datatype arg* pair can be preceded by REF

Argument	Description
RPCFUNC	A keyword indicating that this declaration is for a stored procedure in a DBMS, not an external function in a DLL FOR INFO For information on declaring external functions, see "Declaring external functions" on page 61
ALIAS FOR *"spname"* (optional)	Keywords followed by a string naming the procedure in the database. If the name in the database 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 *"spname"* to establish the association between the PowerScript name and the database name

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 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 INFO For more information, see *Application Techniques*.

Examples

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

```
FUNCTION double GIVE_RAISE(ref double SALARY) RPCFUNC
ALIAS FOR "GIVE_RAISE_PROC"
```

This code calls the function in a script:

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

Example 2 This declaration for the stored procedure smp8 doesn't need an ALIAS FOR phrase, because the PowerBuilder and DBMS names are the same:

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

This code calls the SPM8 stored procedure:

```
int myresult
myresult = SQLCA.spm8(myresult)
```

```
IF SQLCA.sqlcode <> 0 THEN
   messagebox("Error", SQLCA.sqlerrtext)
END IF
```

CHAPTER 4 Operators and Expressions

About this chapter

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

Contents

Topic	Page
Operators	74
Operator precedence in expressions	79
Data type of expressions	80

Operators

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 data types
- Relational operators—for all data types
- Concatenation operators—for string data types

Arithmetic operators

Description

These are the arithmetic operators:

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

Usage

Operator shortcuts for assignments For information about shortcuts that combine arithmetic operators with assignments (such as ++ and +=), see Assignment on page 128.

Subtraction If the option Allow Dashes in Identifiers is checked in the PowerScript painter, you must always surround the subtraction operator and the -- operator with spaces. Otherwise, PowerBuilder interprets the expression as an identifier.

FOR INFO For information about dashes in identifiers, see "Identifier names" on page 6.

Multiplication and division Multiplication and division are carried out to full precision (16–18 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 INFO For more information about NULL values, see "NULL values" on page 11.

Errors and overflows Division by zero, exponentiation of negative values, and so on cause errors during execution.

Overflow of real, double, and decimal values causes errors during execution. 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 INFO For more information about type promotion, see "Data type of expressions" on page 80.

Examples

Subtraction This statement always means subtract B from A:

```
A - B
```

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

```
A-B
```

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

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

Calculations with NULL When the value of variable c is NULL, the following assignment statements all set the variable a to NULL:

```
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:

```
integer i
i = 32767
i = i + 1      // i is now -32768
```

Relational operators

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.

These are the relational and logical operators:

Operator	Meaning	Example
=	Equals	if Price=100 then Rate=.05
>	Greater than	if Price>100 then Rate=.05
<	Less than	if Price<100 then Rate=.05
<>	Not equal	if Price<>100 then Rate=.05
>=	Greater than or equal	if Price>=100 then Rate=.05
<=	Less than or equal	if Price<=100 then Rate=.05
NOT	Logical negation	if NOT Price=100 then Rate=.05
AND	Logical and	if Tax>3 AND Ship <5 then Rate=.05
OR	Logical or	if Tax>3 OR Ship<5 then Rate=.05

Usage

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

FOR INFO For information on comparing strings regardless of case, see the functions Upper on page 1462 and Lower on page 894.

FOR INFO 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 1154, LeftTrim on page 868, and Trim on page 1448.

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 INFO For more information about NULL values, see "NULL values" on page 11.

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:

```
City1="Austin"
City2="AUSTIN"
IF City1=City2 ...                 // Will return FALSE

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

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

```
City1="Austin"
City2="Austin       "
IF City1=City2 ...              // Will return FALSE
```

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

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

Concatenation operator

Description

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

This is the concatenation operator:

Operator	Meaning	Example
+	Concatenate	"cat " + "dog"

Examples

Example 1 These examples concatenate several strings:

```
string Test
Test = "over" + "stock" // Test contains "overstock"
string Lname, Fname, FullName
FullName = Lname + ', ' + Fname
      // FullName contains last name and first name,
      // separated by a comma and space.
```

Example 2 This example shows how a blob can act as an accumulator when reading data from a file:

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

Operator precedence in 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:

Operator	Purpose
( )	Grouping (see note below on overriding)
+, -	Unary plus and unary minus
^	Exponentiation
*, /	Multiplication and division
+, -	Addition and subtraction; string concatenation
=, >, <, <=, >=, <>	Relational operators
NOT	Negation
AND	Logical and
OR	Logical or

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.

Data type of expressions

General information

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

Three types

There are three types:

- Numeric data types
- String and char data types

Numeric data types

General information

All numeric data types are compatible with each other.

What PowerBuilder does

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

Data type promotion when evaluating numeric expressions

Order of precedence

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

Data type	Precedence
Double	Highest
Real	
Decimal	
UnsignedLong, long	
UnsignedInteger, integer	Lowest

Rules for type promotion

Data types of operands If operands in an expression have different data types, the value whose type has lower precedence is converted to the data type 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.

Operators The effects of operators on an expression's data type 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 data type is, at a minimum, long. When operands have data types of higher precedence, other operands are promoted to match based on the *Data types 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 data type is double.
- **Relational** Relational operators do not cause promotion of numeric types.

Data types of literals

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

The data type of	Is
Integer literals (no decimal point or exponent) within the range of longs	Long
Integer literals beyond the range of longs	UnsignedLong
Numeric literals with a decimal point (but no exponent)	Decimal
Numeric literals with a decimal point and explicit exponent	Double

Out of range
Integer literals beyond the range of UnsignedLong cause compiler errors.

Assignment

General information

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

```
c = a + b
```

the data type of a+b is determined by the data types of a and b. Then the result is converted to the data type of c.

Overflow on assignment

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:

```
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:

```
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 data types

General information

There is no explicit char literal type.

String literals convert to type char using the following rules:

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

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

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

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

  ```
  char c = "~n"
  ```

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

Assigning strings to char arrays

As with other data types, 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:

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

promotes the contents of c to a string before comparison with the string "x".

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

Topic	Page
About structures	86
About objects	88
Assignment for objects and structures	93

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 data type, including standard and object data types 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 INFO 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
```

Referring 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
```

This 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
```

This 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 **instantiate**s 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.

System objects versus user objects

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

System objects The PowerBuilder system objects or classes are inherited from a base class PowerObject. The system classes are the ancestors of all the objects you define. To see the system class hierarchy, look at the System tab in the Browser.

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, your main definition is inherited from the window or user object class. The controls you use are also inherited from the system class for that control.

About user objects

Two types

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

Visual user objects

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

Visual user objects	Description
Standard	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
Custom	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 and the controls they are inherited from plus any that you add
External	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

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:

Class user objects	Description
Standard	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
Custom	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

FOR INFO For information on defining and using user objects, see the *PowerBuilder User's Guide.*

Instantiating objects

Classes versus instances

Because of the way object classes and instances are named, it's 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, w_main is a global variable of type w_main:

```
Open(w_main)
```

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:

```
w_main w_1, w_2
Open(w_1)
Open(w_2)
```

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

```
window w_1, w_2
Open(w_1, "w_main")
Open(w_2, "w_main")
```

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

```
uo_emp_data uo_1, uo_2
uo_1 = CREATE uo_emp_data
uo_2 = CREATE uo_emp_data
```

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 INFO For more information about object variables and assignment, see "User objects that behave like structures" on page 92.

Using ancestors and descendants

Descendent objects

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 don't 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.

Managing memory

What you do

Object instances use memory. So when you are finished with an object, you need to release its memory by destroying the object.

Two ways to do it

Closing a window or a visual user object with the appropriate Close function releases its memory and destroys all the controls it contain:

```
Close(w_1)
```

The DESTROY statement releases memory for class user objects:

```
uo_1 = CREATE uo_emp_data
...
DESTROY uo_1
```

Memory leaks

A variable that holds an object reference has a scope. If the variable goes out of scope, you no longer have access to the object. For example, a local variable goes out of scope when its script is finished. An instance variable, which can be an object reference, goes out of scope when its object is destroyed. If these variables go out of scope before their instantiated object is destroyed, you will not be able to release that memory until the application terminates.

User objects that behave like structures

General information

A nonvisual user object can provide functionality similar to that of a structure. Its instance variables form a collection similar to the variables for the structure. In scripts, you use dot notation to refer to the user object's instance variables, just as you do for structure variables.

Advantages of user objects

The user object can include functions and its own structure definitions, and it allows you to inherit from an ancestor class. None of this is possible with a structure definition.

Memory allocation differences

Memory allocation is different for user objects and structures. An object variable is a reference to the object. Declaring the variable does not allocate memory for the object. After you declare it, you must instantiate it with a CREATE statement. Assignment for a user object is also different (described in the next section).

Autoinstantiated objects

If you want a user object that has methods and inheritance but want the memory allocation of a structure, you can define an autoinstantiated object.

You do not have to create and destroy autoinstantiated objects. Like structures, they are created when they are declared and destroyed when they go out of scope. However, because assignment for autoinstantiated objects behaves like structures, the copies made of the object can be a drawback.

To make a custom class user object autoinstantiated, select Autoinstantiate from the popup menu of the User Object painter.

Assignment for objects and structures

General information

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:

```
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:

```
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.

Restriction on assignment 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:

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

FOR INFO For information about passing structures as function arguments, see "Passing arguments to functions and events" on page 112.

Assignment for objects

Declaring an object variable declares an object reference:

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

Using the CREATE statement creates an instance of the object:

```
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:

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

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

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

Declare variables of the ancestor type:

```
uo_emp_data uo_emp1, uo_emp2
```

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

```
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:

```
uo_emp2 = uo_emp1
```

FOR INFO For information about passing objects as function arguments, see "Passing arguments to functions and events" on page 112.

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 uo_emp1, uo_emp2 // Two object instances
```

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

```
uo_emp1 = uo_emp2
```

You never have multiple references to an autoinstantiated user object.

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

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

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

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

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

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

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

```
uo_empdata uo_emp1 // Ancestor
uo_emp_active uo_empa, uo_empb // Descendants
uo_emp_inactive uo_empi // Another descendant
```

Example of rule 1 When assigning one instance to another from the user objects declared above, some assignments are not allowed by the compiler:

```
uo_empb = uo_empa // Allowed, same type
uo_empa = uo_empi // 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 INFO For more information about passing arguments to functions and events, see "Passing arguments to functions and events" on page 112.

CHAPTER 6 Calling Functions and Events

About this chapter

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

Contents

Topic	Page
About functions and events	98
Syntax for calling functions and events	117
Calling functions and events in an object's ancestor	121

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.

These are the different types of functions and events:

Category	Item	Definition
Events	Event	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
	Userevent	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 INFO For information about defining user events, see the *PowerBuilder User's Guide*
	System or built-in event	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 don't have a return value

Category	Item	Definition
Functions	Function	A program or routine that performs specific processing
	System function	A built-in PowerScript function that is not associated with an object
	Object function	A function that is part of an object's definition. PowerBuilder has many predefined object functions and you can define your own
	User-defined function	A function you define. You define a global functions in the Function painter and object functions in the Window or User Object painter
	Global function	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)
	Local external function	An external function that belongs to an object. You declare it in the Window or User Object painter. Its definition is in another DLL
	Global external function	An external function that you declare in any painter, making it globally accessible. Its definition is in another DLL
	Remote procedure call(RPC)	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

Functions versus events

Functions and events have several similarities as well as a many differences:

Similarities	Differences
Both functions and events have arguments and return values	Events are only associated with objects. Functions can be global or part of an object's definition
You can call object functions and events dynamically or statically (Global or system functions can not be called dynamically)	PowerBuilder uses a different search order when looking for an event versus a function

Similarities	Differences
You can post or trigger the function or event call	A call to an undefined event will not trigger an error, but a call to an undefined function will
	Functions can be overloaded whereas events cannot
	When you define a function, you can restrict access to it. You cannot add scope restrictions when you define events
	You can easily extend or override ancestor events when you are writing scripts in a descendant object. To override a function, you must redefine it. To extend the ancestor function, you can call it in the descendant's version of the function

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 will make. Because there is no performance difference, the decision will be 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 will want to choose one approach for handling most situations.

Finding and executing functions and events

General information

PowerBuilder looks for a matching function or event based on its name and its argument list. PowerBuilder can make a match between compatible data types (such as all the numeric types)—the match doesn't have to be exact. PowerBuilder ranks compatible data types to quantify how closely one data type 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 109.

Unqualified function names If you don't 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

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

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

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

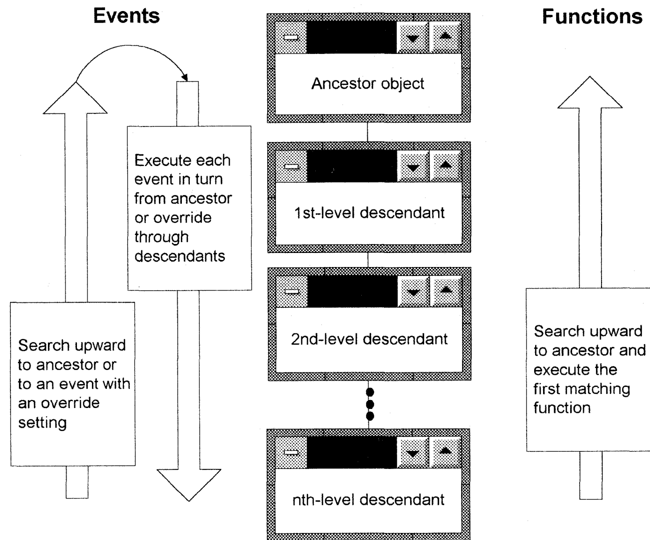
FOR INFO For the syntax for calling ancestor functions, see "Calling functions and events in an object's ancestor" on page 121.

Finding events

PowerBuilder events in descendant 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 descendant event.

Finding functions versus events

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



Triggering versus posting functions and events

Triggering

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.But if other system events have occurred in the meantime, its position in the queue may 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).

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 will cause a compiler error. Both uses require a return value:

```
IF POST IsNull( ) THEN ...
w_1.uf_getresult(dw_1.POST GetBorderStyle(2))
```

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

Examples of posting The following examples illustrate how to post events.

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

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

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

```
parent.POST uf_process_item ()
```

Static versus dynamic calls

Calling functions and events

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

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

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

Static is the default

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 descendant has a function that overrides the ancestor's function (the one found at compile time), the function in the descendant is executed.

Dynamic calls

You specify dynamic calls

When you specify a dynamic call, the function or event does not have to exist when you compile the code. You are saying to the compiler: trust me—there will be a suitable function or event available at execution time.

For a dynamic call, PowerBuilder waits until it's 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 don't exist in the ancestor.

Results of dynamic calls

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

- Ancestor window w_a with a function Set(integer)
- Descendent window w_a_desc
  - Set(integer) overrides the ancestor function
  - Set(string) overload of the function

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

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

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

This statement	Has this result
`mywindow.Set(1)`	Compiles correctly because function is found in the ancestor w_a At execution time, Set(integer) in the *ancestor* is executed
`mywindow.Set("hello")`	Fails to compile; no function prototype in w_a matches the call
`mywindow.DYNAMIC Set("hello")`	Compiles successfully because of the DYNAMIC keyword An error occurs at execution time because no matching function is found

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

```
w_a mywindow
Open(mywindow, "w_a_desc")
```

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

This statement	Has this result
`mywindow.Set(1)`	Compiles correctly because function is found in the ancestor w_a At execution time, Set(integer) in the *descendant* is executed
`mywindow.Set("hello")`	Fails to compile; no function prototype in the ancestor matches the call
`mywindow.DYNAMIC Set("hello")`	Compiles successfully because of the DYNAMIC keyword At execution time, Set(string) in the *descendant* is executed

Disadvantages of dynamic calls

Slower performance Because dynamic calls are resolved at execution time, 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 may be unavailable at execution time. *So do not use dynamic calls when a static call will suffice.*

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:

```
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 is run, the value of variable ish_currentsheet is a descendent window that does have a definition for wf_export:

```
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. And although events can exist without a script, if a function is defined it has to have code.

For purposes of illustration, the table below refers to these statements:

Statement 1 This statement calls an function without looking for a return value:

```
object.DYNAMIC funcname( )
```

Statement 2 This statement looks for an integer return value:

```
int li_int
li_int = object.DYNAMIC funcname( )
```

Statement 3 This statement looks for an Any return value:

```
any la_any
la_any = object.DYNAMIC funcname( )
```

If	And	This occurs	Example
The function doesn't exist		Execution error 65: Dynamic function not found	All the statements cause error 65
The function is found and executed but is not defined with a return value	The code is looking for a return value	Execution error 63: Function/event with no return value used in expression	Statements 2 and 3 cause error 63

Events

Events The table below describes different error conditions and refers to these statements:

Statement 1 This statement calls an event without looking for a return value:

```
object.EVENT DYNAMIC eventname( )
```

Statement 2 This example looks for an integer return value:

```
int li_int
li_int = object.EVENT DYNAMIC eventname( )
```

Statement 3 This example looks for an Any return value:

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

If	And	This occurs	Example
The event doesn't exist	The code *is not* looking for a return value	Nothing; the call fails silently	Statement 1 fails but doesn't cause an error
	The code *is* looking for a return value	A NULL of the Any data type is returned	La_any is set to NULL in statement 3
		If the expected data type is not Any, execution error 19 occurs: Cannot convert Any in Any variable to datatype	The assignment to li_int causes execution error 19 in statement 2
The event is found but is not implemented (there is no script)	The event *has* a defined return value	A NULL of the defined data type is returned	If eventname is defined to return integer, li_int is set to NULL in statement 2
	The event *does not have* a defined return value	A NULL of the Any data type is returned	La_any is set to NULL in statement 3
		If the expected data type is not Any, execution error 19 occurs: Cannot convert Any in Any variable to datatype	The assignment to li_int causes execution error 19 in statement 2
The event is found and executed but is not defined with a return value	The code is looking for a return value	Execution error 63: Function/event with no return value used in expression	Statements 2 and 3 cause error 63

When an error occurs When an execution error occurs, you can handle the error in the SystemError event, but you should not allow the application to continue—the SystemError event should clean up and halt the application. If you don't handle the error, the application is terminated.

If the arguments don't match Function arguments are part of the function's definition. Therefore, if the arguments don't match (a compatible match, not an exact match), it is essentially a different function. The result is the same as if the function didn't exist.

If you call an event dynamically and the arguments don't 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. Short of that, your design and testing can ensure that there will always be an appropriate function or event with the correct return data type.

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 data type 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.

```
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

Functions

When functions are inherited, you can choose to overload or override the function definition.

Events

When events are inherited, the scripts for those events are extended by default. You can choose to extend or override the script.

Overloading and overriding functions

General information

Overloading means defining more than one function with the same name but different argument lists. The additional function definitions can be in the same object or a descendant of that object. PowerBuilder compares the argument list of the function call and the function prototype to determine which function to call.

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 (::).

You can overload or override object functions only.

To create

To create an overloaded or overriding function, you declare the function as you would any function (in the Window painter, for example, Declare>Window Functions). The function name and argument list determine whether the function overloads or overrides an existing function.

Type promotion when matching arguments for overloaded functions

Good overloading 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 data types is a good idea and can provide needed functionality for your application.

Problematic overloading If different versions of a function have arguments of related data types (different numeric types or strings and chars), you must consider how PowerBuilder promotes data types 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 data type of the expression may not be obvious. However, the data type is important in determining what version of an overloaded function will be called.

Because of the intricacies of type promotion for numeric data types, you may decide that you should not define overloaded functions with different numeric data types. Changes someone makes later can affect the application more drastically than expected if the change causes a different function to be called.

How type promotion works When PowerBuilder evaluates an expression, it converts the data types of constants and variables so that it can process or combine them correctly.

Numbers When PowerBuilder evaluates numeric expressions, it promotes the data types of values according to the operators and the data types of the other operands. For example, the data type of the expression n/2 is double because it involves division—the data type of n doesn't matter.

Strings When evaluating an expression that involves chars and strings, PowerBuilder promotes chars to strings.

FOR INFO For more information on type promotion, see "Data type of expressions" on page 80.

Using conversion functions You can take control over the data types of expressions by calling a conversion function. The conversion function ensures that the data type of the expression matches the function prototype you want to call.

For example, because the expression n/2 involves division, the data type 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:

```
CalculateHalf(Long(n/2))
```

Extending and overriding events

General information

When you are writing 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.

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

To select extending or overriding, select the script in the PowerScript painter and choose Compile>Extend Ancestor Script or Compile>Override Ancestor Script.

Passing arguments to functions and events

Three ways

Arguments for built-in or user-defined functions and events can be passed three ways:

Method of passing	Description
By value	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
By reference	A pointer to the variable is passed to the function or event script. Changes affect the original variable in the calling script
Read-only	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 data types because it does not create a copy of the data, as with by value. Data types for which read-only provides a performance advantage are strings, blobs, date, time, and DateTime For other data types, read-only provides documentation for other developers by indicating something about the purpose of the argument

Passing arguments in distributed applications
You can pass arguments by value, by reference, and as read-only to remote objects in distributed applications. However, the behavior is somewhat different in a distributed application than in a non-distributed application.For more information about passing arguments in distributed applications, see *Application Techniques*.

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 will 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 won't exist when the function executes. To use information from the SingleLineEdit, you must pass the information itself, such as the object's text, rather than the object.

When passing an object, you never get another copy of the object. By reference and by value affect the object reference, not the object itself.

Objects passed by value Here you pass a copy of the reference to the object. That reference is still pointing to the original object. If you change properties of the object, you are changing the original object. However, you can change the value of the variable so that it points to another object without affecting the original variable.

Objects passed by reference Here you pass a pointer to the original reference to the object. Again, if you change properties of the object, you are changing the original object. You can change the value of the variable that was passed, but the result is different—the original reference now points to the new object.

Objects passed as read-only Here you pass an object as read-only, and you get a copy of the reference to the object. You cannot change the reference to point to a new object (because read-only is equivalent to a CONSTANT declaration), but you *can* change properties of the object.

Passing structures

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

Structures passed by value Here PowerBuilder passes a copy of the structure. You can modify the copy without affecting the original.

Structures passed by reference Here PowerBuilder passes a reference to the structure. When you changes values in the structure, you are modifying the original. You will not get a null object reference, because structures always exist until they go out of scope.

Structures passed as read-only Here you pass a structure as read-only, and PowerBuilder passes a copy of the structure. You cannot modify any members of the structure.

Passing arrays

When an argument is an array, you specify brackets as part of the argument name in the declaration for the function or event.

Variable-size array as an argument 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 will 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:

```
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:

```
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:

```
integer a[10]
```

You call the function the same way, without brackets:

```
uf_convertarray(a)
```

If the array dimensions don't 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 INFO For more information, see "Declaring arrays" on page 51.

Using return values of functions and events

Functions

All built-in PowerScript functions return a value. You can use the return value or ignore it.

To use the return value, assign it to a variable of the appropriate data type or call the function wherever you can use a value of that data type.

Exceptions
If you post a function, you cannot use its return value.

User-defined functions and external functions may or may not return a value.

Examples The built-in Asc function takes a string as an argument and returns the ASCII value of the string's first character:

```
string S1 = "Carton"
int Test
```

```
Test=32+Asc(S1)     // Test now contains the value
                    // 99 (the ASCII 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:

```
dw_1.SelectRow(dw_1.GetRow(), TRUE)
```

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

```
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.

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

General information

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 can't check their return values and make sure they succeeded. If you want to check return values (and checking is always a good idea), you should call each function separately and assign their 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 don't return an object that could be used by the next call.

An event you have defined can have a return value whose data type is an object. You can include such events in a cascaded call.

Syntax for calling functions and events

Description

This syntax is used to call 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* })

Argument	Description
objectname (optional)	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 INFO For the rules PowerBuilder uses to find unqualified function names, see "Finding and executing functions and events" on page 100
type (optional)	A keyword specifying whether you are calling a function or event. Values are: ♦ FUNCTION (Default) ♦ EVENT
calltype (optional)	A keyword specifying when PowerBuilder looks for the function or event. Values are: ♦ STATIC (Default) ♦ DYNAMIC FOR INFO For more information about static versus dynamic calls, see "Static versus dynamic calls" on page 103 For more information on dynamic calls, see "Dynamic calls" on page 105

Argument	Description
when (optional)	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 INFO For more about triggering and posting, see "Triggering versus posting functions and events" on page 102
name	The name of the function or event you want to call
argumentlist (optional)	The values you want to pass to *name*. Each value in the list must have a data type that corresponds to the declared data type in the function or event definition or declaration

Usage

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's 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, you can use the ancestor operator (::).

FOR INFO For more information about the scope operator for ancestors, see "Calling functions and events in an object's ancestor" on page 121.

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 INFO For more information about cascaded calls, see "Using cascaded calling and return values" on page 115.

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 61.

Case insensitivity
Function and event names are not case sensitive. For example, the following three statements are equivalent:

```
Clipboard("PowerBuilder")
clipboard("PowerBuilder")
CLIPBOARD("PowerBuilder")
```

The PowerBuilder documentation shows built-in functions with uppercase letters for the first character of each word in the function name, such as MessageBox.

Examples

Example 1 The following statements show various function calls using the most simple construction of the function call syntax.

This statement calls the system function Asc:

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

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

```
Update( )
```

This statement calls the global user-defined function gf_setup_appl:

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

This statement calls the system function PrintRect:

```
PrintRect(job, 250, 250, 7500, 1000, 50)
```

Example 2 The following statements show calls to global and system functions.

This statement posts the global user-defined function gf_setup_appl. The function is executed when the calling script finishes:

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

This statement posts the system function PrintRect. It is executed when the calling script finishes. The print job specified in job must still be open:

```
POST PrintRect(job, 250, 250, 7500, 1000, 50)
```

Example 3 In a script for a control, these statements call a user-defined function defined in the parent window. The statements are equivalent, because FUNCTION, STATIC, and TRIGGER are the defaults:

```
Parent.FUNCTION STATIC TRIGGER wf_process( )
Parent.wf_process( )
```

Example 4 This statement in a DataWindow control's Clicked script calls the DoubleClicked event for the same control. The arguments the system passed to Clicked are passed on to DoubleClicked. When triggered by the system, PowerBuilder passes DoubleClicked those same arguments:

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

This statement posts the same event:

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

Example 5 The variable iw_a is an instance variable of an ancestor window type w_ancestorsheet:

```
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:

```
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

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 :: operator to call the ancestor's version explicitly.

Syntax

{ *objectname.* } *ancestorclass* ::{ *type* } { *when* } *name* ({ *argumentlist* })

Argument	Description
objectname (optional)	The name of the object whose ancestor contains the function you want want to execute
ancestorclass	The name of the ancestor class whose function or event you want to execute. The pronoun Super provides the appropriate reference when *ancestorobject* is the immediate ancestor of the current object
type (optional)	A keyword specifying whether you are calling a function or event. Values are: ♦ (Default) FUNCTION ♦ EVENT
when (optional)	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
name	The name of the object function or event you want to call
argumentlist (optional)	The values you want to pass to *name*. Each value in the list must have a data type that corresponds to the declared data type in the function definition

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 data type of the *AncestorReturnValue* variable is always the same as the data type 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, the Script painter 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.

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 the CALL syntax:

CALL SUPER::event_name

or

CALL ancestor_name::event_name

The compiler cannot differentiate between the keyword SUPER and the name of the ancestor. The keyword is replaced with the name of the ancestor before the script is compiled.

The *AncestorReturnValue* variable is only declared and a value assigned when you use the CALL event syntax. It is not declared if you use the new event syntax:

ancestor_name::EVENT event_name( )

You can use the same code in a script that overrides its ancestor event script, but you must insert a CALL statement before you use the *AncestorReturnValue* variable.

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

FOR INFO For information about CALL, see CALL on page 131.

Examples

Example 1 Suppose a window w_ancestor has an event ue_process. A descendent window has a script for the same event.

This statement in a script in the descendant searches the event chain and calls all appropriate events. If the descendant extends the ancestor script, it calls a script for each ancestor in turn followed by the descendent script. If the descendant overrides the ancestor, it calls the descendent script only:

```
EVENT ue_process( )
```

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

```
w_ancestor::EVENT ue_process( )
```

Example 2 You can use the pronoun 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.

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

Example 3 These statements call a function wf_myfunc in the ancestor window (presumably, the descendant also has a function called wf_myfunc):

```
Super::wf_myfunc( )
Super::POST wf_myfunc( )
```

Example 4 Suppose an ancestor window specifies a return code in the CloseQuery event to control whether the window will be closed. If the descendent object also has a script for the event, the ancestor's return code does not get returned to the system. To get the return code, override the ancestor event and include this code in the descendant to trigger it:

```
result = Super::EVENT CloseQuery( )
RETURN result
```

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

Topic	Page
Assignment	128
CALL	131
CHOOSE CASE	132
CONTINUE	134
CREATE	135
DESTROY	139
DO...LOOP	140
EXIT	143
FOR...NEXT	144
GOTO	146
HALT	147
IF...THEN	148
RETURN	151

Assignment

Description

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

Syntax

variablename = expression

Argument	Description
variablename	The name of the variable or object property to which you want to assign a value. *Variablename* can include dot notation to qualify the variable with one or more object names
expression	An expression whose data type is compatible with *variablename*

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:

```
String1 = "Part is out of stock"
TaxRate = .05
```

No multiple assignments Since the equal sign is also a logical operator, you cannot assign more than one variable in a single statement. For example, the following statement does not assign the value 0 to A and B:

```
A=B=0    // This will not assign 0 to A and B.
```

This statement first evaluates B=0 to TRUE or FALSE and then tries to assign this boolean value to A. When A is not a boolean variable, this line produces an error when compiled.

Assigning array values You can assign multiple array values with one statement, such as:

```
int Arr[]
Arr = {1, 2, 3, 4}
```

You can also copy array contents. For example:

```
Arr1 = Arr2
```

copies the contents of Arr2 into array Arr1.

Operator shortcuts These PowerScript shortcuts for assigning values to variables have slight performance advantages over their equivalents:

Assignment	Example	Equivalent to
++	i ++	i = i + 1
--	i --	i = i - 1
+=	i += 3	i = i + 3
-=	i -= 3	i = i -3
*=	i *= 3	i = i * 3
/=	i /= 3	i = i / 3
^=	i ^=3	i = i ^ 3

Unless you have prohibited the use of dashes in variable names, you must leave a space before -- and -= (otherwise, PowerScript reads the minus sign as part of a variable name).

FOR INFO For more information, see "Identifier names" on page 6.

Examples

Example 1 These statements each assign a value to the variable ld_date:

```
date ld_date
ld_date = Today( )
ld_date = 1996-01-01
ld_date = Date("January 1, 1996")
```

Example 2 This statement assigns the parent of the current control to a window variable:

```
window lw_current_window
lw_current_window = Parent
```

Example 3 This statement makes a CheckBox invisible:

```
cbk_on.Visible = FALSE
```

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

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

Example 5 These statements concatenate two strings and assigns the value to the string Text1:

```
string Text1
Text1 = sle_emp.Text+".DAT"
```

Operator shortcuts These assignments use operator shortcuts:

```
int i = 4
i ++          // i is now 5.
i --          // i is 4 again.
i += 10       // i is now 14.
i /= 2        // i is now 7.
```

These shortcuts can be used only in pure assignment statements. They cannot be used with other operators in a statement. For example, the following is invalid:

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

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

```
int i, j
i = 12
i ++
j = I
```

CALL

Description

Calls an ancestor script from a script for a descendent object. You can call scripts for events in an ancestor of the user object, menu, or window. You can also call scripts for events for controls in an ancestor of the user object or window.

When you use the CALL statement to call an ancestor event script, the *AncestorReturnValue* variable is generated. For more information on the *AncestorReturnValue* variable, see "About events" on page 196.

Syntax

CALL *ancestorobject* {`*controlname*}::*event* { (*argumentlist*) }

Parameter	Description
ancestorobject	An ancestor of the descendent object
controlname (optional)	The name of a control in an ancestor window or custom user object
event	An event in the ancestor object
argumentlist (optional)	The values you want to pass to *event* If you do not specify arguments and the event has arguments defined for it, NULL values of the appropriate data type will be passed instead

Usage

New syntax New syntax for calling functions and events allows you to trigger or post an event or function in an ancestor and then pass arguments. But the new syntax does not allow you to call a script for a control in the ancestor.

FOR INFO For more information about the new syntax, see "Calling functions and events in an object's ancestor" on page 121.

Super In some circumstances, you can use the pronoun Super when *ancestorobject* is the descendant object's immediate ancestor.

FOR INFO See the discussion of "Super" on page 17.

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 INFO For information on how to break out of the loop, see EXIT on page 143.

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:

```
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:

```
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 which 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):

objectvariable = CREATE *objecttype*

Parameter	Description
objectvariable	A global, instance, or local variable whose data type is *objecttype*
objecttype	The object data type

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

objectvariable = CREATE USING *objecttypestring*

Parameter	Description
objectvariable	A global, instance, or local variable whose data type is the same class as the object being created or an ancestor of that class
objecttypestring	A string whose value is the name of the class data type to be created

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 1035.)

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 data type, 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 via 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 are finished with an object you created, you can call DESTROY to release its memory.However, you should only call DESTROY 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 INFO For more information about garbage collection, see *Application Techniques*.

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 may or may not exist, the code that accesses it checks whether it exists before calling its functions.

The object that creates the service object declares invo_service as an instance variable:

```
n_service invo_service
```

The Open event for the object creates the service object:

```
//Open event of some object
IF (some condition) THEN
   invo_service = CREATE n_service
END IF
```

When another script wants to call a function that belongs to the n_service class, it checks that invo_service is instantiated:

```
IF IsValid(invo_service) THEN
invo_service.of_perform_some_work( )
END IF
```

If the service object was created, then it also needs to be destroyed:

```
IF isvalid(invo_service) THEN DESTROY invo_service
```

Example 3 When you create a DataStore object, you also have to give it a DataObject and call SetTransObject before you can use it:

```
l_ds_delete = CREATE u_ds
l_ds_delete.DataObject = 'd_user_delete'
l_ds_delete.SetTransObject(SQLCA)
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_win16 and n_file_service_class_win32 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 Windows 3.1:

```
n_file_service_class  lnv_fileservice
string ls_objectname
environment luo_env

GetEnvironment ( luo_env )
IF luo_env.Win31 = TRUE THEN
   ls_objectname = "n_file_service_class_win16"
```

```
ELSE
    ls_objectname = "n_file_service_class_win32"
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 data type 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 only call DESTROY 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 INFO For more information about garbage collection, see *Application Techniques*.

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

Parameter	Description
condition	The condition you are testing
statementblock	The block of statements you want to repeat

DO WHILE *condition*
statementblock
LOOP

Parameter	Description
condition	The condition you are testing
statementblock	The block of statements you want to repeat

DO
statementblock
LOOP UNTIL *condition*

Parameter	Description
statementblock	The block of statements you want to repeat
condition	The condition you are testing

DO
statementblock
LOOP WHILE *condition*

Parameter	Description
statementblock	The block of statements you want to repeat
condition	The condition you are testing

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:

```
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:

```
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:

```
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:

```
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 INFO For information on how to jump to the end of the loop and continue looping, see CONTINUE on page 134.

Examples

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

```
int Nbr[10]
int Count = 1
// Assume values get assigned to Nbr array...

DO WHILE Count < 11
    IF Nbr[Count] = 0 THEN EXIT
    Count = Count + 1
LOOP

MessageBox("Hi", "Count is now " + String(Count) )
```

Example 2 This EXIT statement causes the loop to terminate if an element in the Nbr array equals 0:

```
int Nbr[10]
int Count
// Assume values get assigned to Nbr array...

FOR Count = 1 to 10
    IF Nbr[Count] = 0 THEN EXIT
NEXT

MessageBox("Hi", "Count is now " + String(Count) )
```

FOR...NEXT

Description

A control structure that is a numerical iteration, used to execute one or more statements a specified number of times.

Syntax

FOR *varname* = *start* TO *end* {STEP *increment*}
    statementblock
NEXT

Parameter	Description
varname	The name of the iteration counter variable. It can be any numerical type (integer, double, real, long, or decimal), but integers provide the fastest performance
start	Starting value of *varname*
end	Ending value of *varname*
increment (optional)	The increment value. *Increment* must be a constant and the same data type as *varname*. If you enter an increment, STEP is required. +1 is the default increment
statementblock	The block of statements you want to repeat

Usage

Using the start and end parameters For a positive *increment*, *end* must be greater than *start*. For a negative *increment*, *end* must be less than *start*.

When *increment* is positive and *start* is greater than *end*, *statementblock* does not execute. When *increment* is negative and *start* is less than *end*, *statementblock* does not execute.

When *start* and *end* are expressions, they are reevaluated on each pass through the loop. So if the expression's value changes, it will affect the number of loops. Consider this example—the body of the loop changes the number of rows, which changes the result of the RowCount function:

```
FOR n = 1 TO dw_1.RowCount( )
   dw_1.DeleteRow(1)
NEXT
```

A variable as the step increment
If you need to use a variable for the step increment, you can use one of the DO...LOOP constructions and increment the counter yourself within the loop.

Nesting You can nest FOR...NEXT statements. You must have a NEXT for each FOR.

You can end the FOR loop with the keywords END FOR instead of NEXT.

Avoid overflow
If *start* or *end* is too large for the data type of *varname*, *varname* will overflow, which could create an infinite loop. Consider this statement for the integer li_int.

```
FOR li_int = 1 TO 50000
```

The end value 50000 is too large for an integer. When li_int is incremented, it overflows to a negative value before reaching 50000, creating an infinite loop.

Examples

Example 1 These statements add 10 to A as long as n is >=5 and <=25:

```
FOR n = 5 to 25
   A = A+10
NEXT
```

Example 2 These statements add 10 to A and increment n by 5 as long as n is >= 5 and <=25:

```
FOR N = 5 TO 25 STEP 5
   A = A+10
NEXT
```

Example 3 These statements contain two lines that will never execute because *increment* is negative and *start* is less than *end*:

```
FOR Count = 1 TO 100 STEP -1
   IF Count < 1 THEN EXIT // These 2 lines
   Box[Count] = 10        // will never execute.
NEXT
```

Example 4 These are nested FOR...NEXT statements:

```
Int Matrix[100,50,200]
FOR i = 1 to 100
   FOR j = 1 to 50
      FOR k = 1 to 200
         Matrix[i,j,k]=1
      NEXT
   NEXT
NEXT
```

GOTO

Description

Transfers control from one statement in a script to another statement that is labeled.

Syntax

GOTO *label*

Parameter	Description
label	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

Examples

Example 1 This GOTO statement skips over the Taxable=FALSE line:

```
Goto NextStep
Taxable=FALSE  //This statement will never
               //execute.
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 script for the Close event for the application and then terminates the application. If there is no script for the Close event at the application level, PowerBuilder immediately terminates the application.

Platform information
On Windows 3.1, you should post the HALT CLOSE statement if it is being called from within the DBError event.

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*}

Parameter	Description
condition	The condition you want to test
action1	The action you want performed if the condition is TRUE. The action must be a single statement on the same line as the rest of the IF statement
action2 (optional)	The action you want performed if the condition is FALSE. The action must be a single statement on the same line as the rest of the IF statement

Syntax 2 (the multiline format):

```
IF condition1 THEN
action1
{ ELSEIF condition2 THEN
    action2
. . . }
{ ELSE
action3 }
END IF
```

Parameter	Description
condition1	The first condition you want to test
action1	The action you want performed if *condition1* is TRUE. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines. At least one action is required
condition2 (optional)	The condition you want to test if *condition1* is FALSE. You can have multiple ELSEIF...THEN statements in an IF...THEN control structure
action2	The action you want performed if *condition2* is TRUE. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines

Parameter	Description
action3 (optional)	The action you want performed if none of the preceding conditions is true. The action can be a statement or multiple statements that are separated by semicolons or placed on separate lines

Usage

You can use continuation characters to place the single-line format on more than one physical line in the script.

You must end a multiline IF...THEN control structure with END IF (which is two words).

Examples

Example 1 This single-line IF...THEN statement opens window w_first if Num = 1; otherwise, w_rest is opened:

```
IF Num = 1 THEN Open(w_first) ELSE Open(w_rest)
```

Example 2 This single-line IF...THEN statement displays a message if the value in the SingleLineEdit sle_State is TX. It uses the continuation character to continue the single-line statement across two physical lines in the script:

```
IF sle_State.text="TX" THEN   &
   MessageBox("Hello","Tex")
```

Example 3 This multiline IF...THEN compares the horizontal positions of windows w_first and w_second. If w_first is to the right of w_second, w_first is moved to the left side of the screen:

```
IF w_first.X > w_second.X THEN
   w_first.X = 0
END IF
```

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

- Beep twice if X equals Y
- Display the Parts listbox and highlight item 5 if X equals Z
- Display the Choose listbox if X is blank
- Hide the Empty button and display the Full button if none of the above conditions is TRUE

```
IF X=Y THEN
   Beep(2)
ELSEIF X=Z THEN
   Show (lb_parts); lb_parts.SetState(5,TRUE)
ELSEIF X=" " THEN
```

```
   Show (lb_choose)
ELSE
   Hide(cb_empty)
   Show(cb_full)
END IF
```

RETURN

Description

Stops the execution of a script or function immediately.

Syntax

RETURN { *expression* }

Parameter	Description
expression	In a function, any value (or expression) you want the function to return. The return value must be the data type specified as the return type in the function

Usage

When a user's action triggers an event and PowerBuilder encounters RETURN in the event script, it terminates execution of that script immediately and waits for the next user action.

When a script calls a function or event and PowerBuilder encounters RETURN in the code, RETURN transfers (returns) control to the point at which the function or event was called.

Examples

Example 1 This script causes the system to beep once; the second beep statement will not execute:

```
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
```

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

Topic	Page
Using SQL in scripts	155
CLOSE Cursor	158
CLOSE Procedure	159
COMMIT	160
CONNECT	161
DECLARE Cursor	162
DECLARE Procedure	163
DELETE	165
DELETE Where Current of Cursor	166
DISCONNECT	167
EXECUTE	168
FETCH	169
INSERT	170
OPEN Cursor	171
ROLLBACK	172
SELECT	173
SELECTBLOB	174
UPDATE	175
UPDATEBLOB	176
UPDATE Where Current of Cursor	178
Using dynamic SQL	179

Topic	Page
Dynamic SQL Format 1	183
Dynamic SQL Format 2	184
Dynamic SQL Format 3	186
Dynamic SQL Format 4	189

Using SQL in scripts

General information

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 INFO For information about embedded SQL, see online Help.

Referencing PowerScript variables in scripts

Wherever constants can be referenced in SQL statements, PowerScript variables preceded by a colon (:) can be substituted. Any valid PowerScript variable can be used.

This INSERT statement uses a constant value:

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

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

```
int   Sal_var
Sal_var = 18900
INSERT INTO EMPLOYEE ( SALARY )
   VALUES ( :Sal_var ) ;
```

Using indicator variables

PowerBuilder supports **indicator variables**, which are used to identify NULL values or conversion errors after a database retrieval. Indicator variables are integers that are specified in the *HostVariableList* of a FETCH or SELECT statement.

Each indicator variable is separated from the variable it is indicating by a space (but no comma). For example, this statement is a *HostVariableList* without indicator variables:

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

The same *HostVariableList* with indicator variables might look like this:

```
:Name :IndVar1, :Address :IndVar2, :City :IndVar3
```

Indicator variables have one of these values:

Numerical value	Meaning
0	Valid, non-NULL value
-1	NULL value

Numerical value	Meaning
-2	Conversion error

Error reporting
Not all DBMSs return a conversion error when the data type of a column does not match the data type of the associated variable.

The following statement uses the indicator variable IndVar2 to see if Address contains a NULL value:

```
if IndVar2 = -1 then...
```

You could also use the PowerScript IsNull function to accomplish the same result without using indicator variables:

```
if IsNull( Address ) then ...
```

This statement uses the indicator variable IndVar3 to set City to NULL:

```
IndVar3 = -1
```

You could also use the PowerScript SetNull function to accomplish the same result without using indicator variables:

```
SetNull( City )
```

FOR INFO For information about the SetNull function, see SetNull on page 1296.

Error handling in scripts

The scripts shown in the SQL examples above do not include error handling, but it is good practice to test the success and failure codes (the SQLCode attribute) in the transaction object after *every* statement. The codes are:

Value	Meaning
0	Success
100	Fetched row not found
-1	Error; the statement failed. Use SQLErrText or SQLDBCode to obtain the detail.

About SQLErrText and SQLDBCode The string SQLErrText in the transaction object contains the database vendor–supplied error message. The long named SQLDBCode in the transaction object contains the database vendor–supplied status code:

```
IF SQLCA.SQLCode = -1 THEN
```

```
MessageBox("SQL error", SQLCA.SQLErrText)
END IF
```

Painting standard SQL

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 INFO For more information about scope, see "Where to declare variables" on page 36.

You can declare cursors and stored procedures at the scope of global, instance, shared, or local variables. A global, instance, or shared cursor or procedure can be declared in the Window, User Object, Menu, or PowerScript painter using the Declare menu bar option. A local cursor or procedure can be declared in the PowerScript painter or Function painter using the Paste SQL button in the PainterBar.

You can paint standard embedded SQL statements in the PowerScript painter, the Function painter, and the Database Administration painter using the Paste SQL button in the PainterBar or the Edit Paste SQL option from the menu bar.

Supported SQL statements

In general, all DBMS-specific features are supported in PowerScript as long as they occur within a PowerScript-supported SQL statement. For example, PowerScript supports DBMS-specific built-in functions within a SELECT command.

CLOSE Cursor

Description

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

Syntax

CLOSE *CursorName* ;

Parameter	Description
CursorName	The cursor you want to close

Usage

This statement must be preceded by an OPEN statement for the same cursor. The USING TransactionObject clause is not allowed with CLOSE; the transaction object was specified in the statement that declared the cursor.

CLOSE often appears in the script that is executed when the SQL code after a fetch equals 100 (not found).

Error Handling
It is good practice to test the success/failure code after executing a CLOSE Cursor statement.

Examples

This statement closes the Emp_cursor cursor:

```
CLOSE Emp_cursor ;
```

CLOSE Procedure

Description

Closes the SQL procedure *ProcedureName*; ends processing of *ProcedureName*.

DBMS-specific
Not all DBMSs support stored procedures.

Syntax

CLOSE *ProcedureName* ;

Parameter	Description
ProcedureName	The stored procedure you want to close

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.

You only need to use CLOSE to close procedures that return result sets. PowerBuilder automatically closes procedures that don't 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.

Syntax

COMMIT {USING *TransactionObject*} ;

Parameter	Description
TransactionObject	The name of the transaction object for which you want to permanently update all database operations since the previous COMMIT, ROLLBACK, or CONNECT. This clause is required only for transaction objects other than the default (SQLCA)

Usage

COMMIT does not cause a disconnect, but it does close all open cursors or procedures. (But note that the DISCONNECT statement in PowerBuilder does issue a COMMIT.)

Error handling
It is good practice to test the success/failure code after executing a COMMIT statement.

Examples

Example 1 This statement commits all operations for the database specified in the default transaction object:

```
COMMIT ;
```

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

```
COMMIT USING Emp_tran ;
```

CONNECT

Description

Connects to a specified database.

Syntax

CONNECT {USING *TransactionObject*} ;

Parameter	Description
TransactionObject	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)

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*} ;

Parameter	Description
CursorName	Any valid PowerBuilder name
SelectStatement	Any valid SELECT statement
TransactionObject	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)

Usage

DECLARE Cursor is a nonexecutable command and is analogous to declaring a variable.

To declare a global, shared, or instance cursor, select Declare>Global Variables, Declare>Instance Variables, or Declare>Shared Variables in the Window, User Object, Menu, or PowerScript painter. To declare a local cursor, click the Paint SQL button in the PainterBar.

FOR INFO For information about global, instance, shared, and local scope, see "Where to declare variables" on page 36.

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.

Syntax

DECLARE *ProcedureName* PROCEDURE FOR
    StoredProcedureName
    @Param1=Value1, *@Param2=Value2*,...
    {USING *TransactionObject*} ;

Parameter	Description
ProcedureName	Any valid PowerBuilder name
StoredProcedureName	Any stored procedure in the database
@Paramn=Valuen	The name of a parameter (argument) defined in the stored procedure and a valid PowerBuilder expression; represents the number of the parameter and value
TransactionObject	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)

Usage

DECLARE Procedure is a nonexecutable command. It is analogous to declaring a variable.

Using SQL Server
In SQL Server, you can use the optional reserved word OUT to indicate an output parameter:

```
@Param=Value OUT
```

When using Sybase SQL Server stored procedures, observe Sybase naming conventions for local variables. Make sure the local variable name has no more than 13 characters, including the @ sign.

To declare a global, shared or instance procedure, select Declare>Global Variables, Declare>Instance Variables, or Declare>Shared Variables in the Window, User Object, Menu, or PowerScript painter. To declare a local procedure, click the Paint SQL button in the PainterBar.

FOR INFO For information about global, instance, shared, and local scope, see "Where to declare variables" on page 36.

Examples

Example 1 This statement declares the Sybase SQL Server 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*} ;

Parameter	Description
TableName	The name of the table from which you want to delete rows
Criteria	Criteria that specify which rows to delete
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 a DELETE statement.

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

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* ;

Parameter	Description
Cursor or Procedure	The name of the cursor or procedure from which you want to fetch a row
HostVariableList	PowerScript variables into which data values will be retrieved

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.

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:

```
int         Emp_num
string      Emp_name
FETCH Emp_cur INTO :Emp_num, :Emp_name ;
```

Example 2 If sle_emp_num and sle_emp_name are SingleLineEdits, these statements fetch from the cursor named Emp_cur, store the data in Emp_num and sle_emp_name, and then convert Emp_num from an integer to a string and put it in sle_emp_num:

```
int         Emp_num
FETCH Emp_cur into :emp_num, :sle_emp_name.Text ;
sle_emp_num.Text = string(Emp_num)
```

INSERT

Description

Inserts one or more new rows into the table specified in *RestOfInsertStatement*.

Syntax

INSERT *RestOfInsertStatement*
{USING *TransactionObject*} ;

Parameter	Description
RestOfInsertStatement	The rest of the INSERT statement (the INTO clause, list of columns and values or source)
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 INSERT statement.

Examples

Example 1 These statements insert a row with the values in EmpNbr and EmpName into the Emp_nbr and Emp_name columns of the Employee table identified in the default transaction object:

```
int EmpNbr
string EmpName
...
INSERT INTO Employee (employee.Emp_nbr,
   employee.Emp_name)
   VALUES (:EmpNbr, :EmpName) ;
```

Example 2 These statements insert a row with the values entered in the SingleLineEdits sle_number and sle_name into the Emp_nbr and Emp_name columns of the Employee table in the transaction object named Emp_tran:

```
int      EmpNbr
EmpNbr = Integer(sle_number.Text)
INSERT INTO Employee (employee.Emp_nbr,
   employee.Emp_name)
   VALUES (:EmpNbr, :sle_name.Text) 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.

Syntax

ROLLBACK {USING *TransactionObject*} ;

Parameter	Description
TransactionObject	The name of the transaction object that identifies the database in which you want to cancel all operations since the last COMMIT, ROLLBACK, or CONNECT. This clause is required only for transaction objects other than the default (SQLCA)

Usage

ROLLBACK does not cause a disconnect, but it does close all open cursors and procedures.

Error handling
It is good practice to test the success/failure code after executing a ROLLBACK statement.

Examples

Example 1 This statement cancels all database operations in the database specified in the default transaction object:

```
ROLLBACK ;
```

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

```
ROLLBACK USING emp_tran ;
```

SELECT

Description

Selects a row in the tables specified in *RestOfSelectStatement*.

Syntax

SELECT *RestOfSelectStatement*
{USING *TransactionObject*} ;

Parameter	Description
RestOfSelectStatement	The rest of the SELECT statement (the column list INTO, FROM, WHERE, and other clauses)
TransactionObject	The name of the transaction object that identifies the database containing the table. This clause is required only for transaction objects other than the default (SQLCA)

Usage

An error occurs if the SELECT statement returns more than one row.

Error handling
It is good practice to test the success/failure code after executing a SELECT statement.

Examples

The following statements select data in the Emp_LName and Emp_FName columns of a row in the Employee table and put the data into the SingleLineEdits sle_LName and sle_FName (the transaction object Emp_tran is used):

```
int   Emp_num
Emp_num = Integer(sle_Emp_Num.Text)

SELECT employee.Emp_LName, employee.Emp_FName
   INTO :sle_LName.text, :sle_FName.text
   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
```

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.

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):

```
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 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*} ;

Parameter	Description
TableName	The name of the table in which you want to update rows
RestOfUpdateStatement	The rest of the UPDATE statement (the SET 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

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:

```
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 data type of this column must be blob
BlobVariable	A PowerScript variable of the data type 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.

Examples

These statements update the blob column emp_pic in the Employee table where emp_num is 100:

```
int    fh
blob   Emp_id_pic
fh = FileOpen("c:\emp_100.bmp", StreamMode!)
IF fh <> -1 THEN
   FileRead(fh, emp_id_pic)
   FileClose(fh)
   UPDATEBLOB Employee SET emp_pic = :Emp_id_pic
      WHERE Emp_num = 100
      USING Emp_tran ;
END IF
```

```
IF Emptran.SQLNRows > 0 THEN
    COMMIT USING Emp_tran ;
END IF
```

The blob Emp_id_pic requires a colon to indicate it is a host (PowerScript) variable in the UPDATEBLOB statement.

UPDATE Where Current of Cursor

Description

Updates the row in which the cursor is positioned using the values in *SetStatement*.

Syntax

UPDATE *TableName SetStatement*
    WHERE CURRENT OF *CursorName* ;

Parameter	Description
TableName	The name of the table in which you want to update the row
SetStatement	The word SET followed by a comma-separated list of the form *ColumnName* = *value*
CursorName	The name of the cursor in which the table is referenced

Usage

The USING Transaction Object clause is not allowed with UPDATE Where Current of Cursor; the transaction object was specified in the statement that declared the cursor.

Examples

This statement updates the row in the Employee table in which the cursor called Emp_curs is positioned:

```
UPDATE Employee
   SET salary = 17800
   WHERE CURRENT of Emp_curs ;
```

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 SQL Anywhere
For information about using dynamic SQL with SQL Anywhere, see the *SQL Anywhere User's Guide*.

Four formats

PowerBuilder has four dynamic SQL formats. Each format handles one of the following situations at compile time:

Format	When used
Format 1	Non-result-set statements with no input parameters
Format 2	Non-result-set statements with input parameters
Format 3	Result-set statements in which the input parameters and result-set columns are known at compile time
Format 4	Result-set statements in which the input parameters, the result-set columns or both are unknown at compile time

To handle these situations, you use:

- The PowerBuilder dynamic SQL statements
- The dynamic versions of CLOSE, DECLARE, FETCH, OPEN, and EXECUTE
- The PowerBuilder data types DynamicStagingArea and DynamicDescriptionArea

About the examples
The examples assume that the default transaction object (SQLCA) has been assigned valid values and that a successful CONNECT has been executed. Although the examples do not show error checking, you should check the SQLCode after each SQL statement.

Dynamic SQL statements

The PowerBuilder dynamic SQL statements are:

```
DESCRIBE DynamicStagingArea
    INTO DynamicDescriptionArea ;
EXECUTE {IMMEDIATE} SQLStatement
    {USING TransactionObject} ;
EXECUTE DynamicStagingArea
    USING ParameterList ;
EXECUTE DYNAMIC Cursor | Procedure
    USING ParameterList ;
OPEN DYNAMIC Cursor | Procedure
    USING ParameterList ;
EXECUTE DYNAMIC Cursor | Procedure
    USING DESCRIPTOR DynamicDescriptionArea ;
OPEN DYNAMIC Cursor | Procedure
    USING DESCRIPTOR DynamicDescriptionArea ;
PREPARE DynamicStagingArea
    FROM SQLStatement {USING TransactionObject} ;
```

Two data types

DynamicStagingArea DynamicStagingArea is a PowerBuilder data type. 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 data type. PowerBuilder uses a variable of this type to store information about the input and output parameters used in Format 4 of dynamic SQL.

PowerBuilder provides a global DynamicDescriptionArea named SQLDA that you can use when you need a DynamicDescriptionArea variable.

If necessary, you can declare and create additional object variables of the type DynamicDescriptionArea. These statements declare and create the variable, which must be done before referring to it in a dynamic SQL statement:

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

FOR INFO For more information about SQLDA, see Dynamic SQL Format 4 on page 189.

Preparing to use dynamic SQL

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 data types 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 data types. However, this is required only when your script requires simultaneous access to two or more dynamically prepared statements.

This is a *valid* dynamic cursor:

```
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
PREPARE SQLSA FROM "SELECT emp_id FROM employee" ;
OPEN DYNAMIC my_cursor ;
```

This is an *invalid* dynamic cursor (there is no PREPARE, and therefore an execution error will occur):

```
DECLARE my_cursor DYNAMIC CURSOR FOR SQLSA ;
OPEN DYNAMIC my_cursor ;
```

Statement order Where you place the dynamic SQL statements in your scripts is unimportant, but the order of execution is important in Formats 2, 3, and 4. You must execute:

1. The DECLARE and the PREPARE before you execute any other dynamic SQL statements
2. The OPEN in Formats 3 and 4 before the FETCH
3. The CLOSE at the end

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

These statements illustrate the correct ordering:

```
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*} ;

Parameter	Description
SQLStatement	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
TransactionObject (optional)	The name of the transaction object that identifies the database

Examples

These statements create a database table named Employee. The statements use the string Mysql to store the CREATE statement.

For SQL Server users
If you are connected to a SQL Server database, set AUTOCOMMIT to TRUE before executing the CREATE.

```
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:

```
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*} ;

Parameter	Description
DynamicStagingArea	The name of the DynamicStagingArea (usually SQLSA) If you need a DynamicStagingArea variable other than SQLSA, you must declare it and instantiate it with the CREATE statement before using it
SQLStatement	A string containing a valid SQL statement. The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions Enter a question mark (?) for each parameter in the statement. Value substitution is positional; reserved word substitution is not allowed
TransactionObject (optional)	The name of the transaction object that identifies the database
ParameterList (optional)	A comma-separated list of PowerScript variables. Note that PowerScript variables are preceded by a colon (:)

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* ;

Parameter	Description
Cursor or *Procedure*	The name of the cursor or procedure you want to use
DynamicStagingArea	The name of the DynamicStagingArea (usually SQLSA) If you need a DynamicStagingArea variable other than SQLSA, you must declare it and instantiate it with the CREATE statement before using it
SQLStatement	A string containing a valid SQL SELECT statement The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions Enter a question mark (?) for each parameter in the statement. Value substitution is positional; reserved word substitution is not allowed
TransactionObject (optional)	The name of the transaction object that identifies the database
ParameterList (optional)	A comma-separated list of PowerScript variables. Note that PowerScript variables are preceded by a colon (:)
HostVariableList	The list of PowerScript variables into which the data values will be retrieved

Usage

To specify a NULL value, use the SetNull function.

The DECLARE statement is not executable and can be declared globally.

If your DBMS supports formats of FETCH other than the customary (and default) FETCH NEXT, you can specify FETCH FIRST, FETCH PRIOR, or FETCH LAST.

The FETCH and CLOSE statements in Format 3 are the same as in standard embedded SQL.

To declare a global, shared, or instance cursor or procedure, select Global Variables, Instance Variables, or Shared Variables on the Declare menu of the PowerScript painter. To declare a local cursor, click the Paint SQL button in the PainterBar.

FOR INFO For information about global, instance, shared, and local scope, see "Where to declare variables" on page 36.

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:

```
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:

```
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:

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

EXECUTE DYNAMIC Cursor | Procedure
    USING DESCRIPTOR DynamicDescriptionArea ;

FETCH Cursor | Procedure
    USING DESCRIPTOR DynamicDescriptionArea ;

CLOSE Cursor | Procedure ;
```

Parameter	Description
Cursor or *Procedure*	The name of the cursor or procedure you want to use
DynamicStagingArea	The name of the DynamicStagingArea (usually SQLSA) If you need a DynamicStagingArea variable other than SQLSA, you must declare it and instantiate it with the CREATE statement before using it
SQLStatement	A string containing a valid SQL SELECT statement. The string can be a string constant or a PowerBuilder variable preceded by a colon (such as :mysql). The string must be contained on one line and cannot contain expressions Enter a question mark (?) for each parameter in the statement. Value substitution is positional; reserved word substitution is not allowed
TransactionObject (optional)	The name of the transaction object that identifies the database

Parameter	Description
DynamicDescriptionArea	The name of the DynamicDescriptionArea (usually SQLDA) If you need a DynamicDescriptionArea variable other than SQLDA, you must declare it and instantiate it with the CREATE statement before using it

Usage

The DECLARE statement is not executable and can be defined globally.

If your DBMS supports formats of FETCH other than the customary (and default) FETCH NEXT, you can specify FETCH FIRST, FETCH PRIOR, or FETCH LAST.

To declare a global, shared, or instance cursor or procedure, select Global Variables, Instance Variables, or Shared Variables on the Declare menu of the PowerScript painter. To declare a local cursor, click the Paint SQL button in the PainterBar.

FOR INFO For information about global, instance, shared, and local scope, see "Where to declare variables" on page 36.

Accessing attribute information When a statement is described into a DynamicDescriptionArea, this information is available to you in the attributes of that DynamicDescriptionArea variable:

Information	Attribute
Number of input parameters	NumInputs
Array of input parameter types	InParmType
Number of output parameters	NumOutputs
Array of output parameter types	OutParmType

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 INFO For information about these functions, see GetDynamicDate on page 669, GetDynamicDateTime on page 671, GetDynamicNumber on page 673, GetDynamicString on page 674, and GetDynamicTime on page 675.

Parameter values The following enumerated data types 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.

Examples

Example 1 These statements assume you know that there will be only one output descriptor and that it will be an integer. You can expand this example to support any number of output descriptors and any data type 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 data type 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. Usage notes and examples provide information about what is typically done in an event's script.

The events are listed in alphabetical order.

About events

There are several types of events:

Type	Occurs in response to
System events with an ID	User actions or other system messages or a call in your scripts
System events without an ID	PowerBuilder messages or a call in your scripts
User-defined events with an ID	User actions or other system messages or a call in your scripts
User-defined events without an ID	A call in your scripts

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.

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 could 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 can't 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.

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 don't 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.

Ancestor event script return values

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.

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 data type of the *AncestorReturnValue* variable is always the same as the data type 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, the Script painter 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.

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 the CALL syntax:

CALL SUPER::event_name

or

CALL ancestor_name::event_name

The compiler cannot differentiate between the keyword SUPER and the name of the ancestor. The keyword is replaced with the name of the ancestor before the script is compiled.

The *AncestorReturnValue* variable is only declared and a value assigned when you use the CALL event syntax. It is not declared if you use the new event syntax:

ancestor_name::EVENT event_name( )

You can use the same code in a script that overrides its ancestor event script, but you must insert a CALL statement before you use the *AncestorReturnValue* variable.

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

FOR INFO For more information about AncestorReturnValue, see *Application Techniques*.

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 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 data type in the Event Declaration dialog box.

The event will never be triggered by user actions or system activity. You will 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 functions and events" on page 117 for information on the calling syntax.

FOR INFO To learn more about user-defined events, see the *PowerBuilder User's Guide*.

Activate

Description

Occurs just before the window becomes active.

Event ID

Event ID	Objects
pbm_activate	Window

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:

```
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:

```
w_sheet.ArrangeSheets(Layer!)
```

See also

Close
Open
Show

BeginDrag

The BeginDrag event has different arguments for different objects:

Object	See
ListView control	Syntax 1
TreeView control	Syntax 2

Syntax 1

For ListView controls

Description

Occurs when the user presses the left mouse button in the ListView control and begins dragging.

Event ID

Event ID	Objects
pbm_lvnbegindrag	ListView

Arguments

Argument	Description
index	Integer by value (the index of the ListView item being dragged)

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.

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

Event ID	Objects
pbm_tvnbegindrag	TreeView

Arguments

Argument	Description
handle	Long by value (handle of the TreeView item being dragged)

Return codes

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

0 Continue processing

Usage

BeginDrag and BeginRightDrag events occur when the user presses the mouse button and drags, whether or not dragging is enabled. To enable dragging, you can:

- Set the DragAuto property to TRUE. If the TreeView's DragAuto property is TRUE, a drag operation begins automatically when the user clicks.
- Call the Drag function. If DragAuto is FALSE, then in the BeginDrag event script, the programmer can call the Drag function to begin the drag operation.

The user cannot drag a highlighted item.

Dragging a TreeView item onto another control causes the control's standard drag events (DragDrop, DragEnter, DragLeave, and DragWithin) to occur. The standard drag events occur for TreeView when another control is dragged within the borders of the TreeView.

Examples

This example moves the first TreeView item in the source TreeView to another TreeView when the user drags there. Itvi_dragged_object is a window instance variable whose type is TreeViewItem. To copy the item, omit the code that deletes it from the source TreeView.

This code is in the BeginDrag event script of the source TreeView:

```
long itemnum

// If the TreeView's DragAuto property is FALSE
This.Drag(Begin!)

itemnum = 1
This.GetItem(itemnum, itvi_dragged_object)

// To copy, rather than move, omit these two lines
This.DeleteItem(itemnum)
This.SetRedraw(TRUE)
```

This code is in the DragDrop event of the target TreeView:

```
This.InsertItemLast(0, ilvi_dragged_object)
This.SetRedraw(TRUE)
```

Instead of deleting the item from the source TreeView immediately, consider deleting it after the insertion in the DragDrop event succeeds.

See also

BeginRightDrag
DragDrop
DragEnter
DragLeave
DragWithin

BeginLabelEdit

The BeginLabelEdit event has different arguments for different objects:

Object	See
ListView control	Syntax 1
TreeView control	Syntax 2

Syntax 1 For ListView controls

Description

Occurs when the user clicks on the label of an item after selecting the item.

Event ID

Event ID	Objects
pbm_lvnbeginlabeledit	ListView

Arguments

Argument	Description
index	Integer by value (the index of the selected ListView item)

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 ListView item being edited:

```
ListViewItem lvi
This.GetItem(index lvi)
sle_info.text = "Editing " + string(lvi.label)
```

See also

EndLabelEdit

Syntax 2 For TreeView controls

Description

Occurs when the user clicks on the label of an item after selecting the item.

Event ID

Event ID	Objects
pbm_tvnbeginlabeledit	TreeView

Arguments

Argument	Description
handle	Long by value (the handle of the selected TreeView item)

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:

```
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:

Object	See
ListView control	Syntax 1
TreeView control	Syntax 2

Syntax 1 For ListView controls

Description

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

Event ID

Event ID	Objects
pbm_lvnbeginrightdrag	ListView

Arguments

Argument	Description
index	Integer by value (the index of the ListView item being dragged)

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.

Examples

See the example for the BeginDrag event. It is effective for the BeginRightDrag event too.

See also

BeginDrag
DragDrop
DragEnter
DragLeave
DragWithin

Syntax 2

For TreeView controls

Description

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

Event ID

Event ID	Objects
pbm_tvnbeginrightdrag	TreeView

Arguments

Argument	Description
handle	Long by value (the handle of the TreeView item being dragged)

Return codes

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

0 Continue processing

Usage

BeginDrag and BeginRightDrag events occur when the user presses the mouse button and drags, whether or not dragging is enabled. To enable dragging, you can:

- Set the DragAuto property to TRUE. If the TreeView's DragAuto property is TRUE, a drag operation begins automatically when the user clicks
- Call the Drag function. If DragAuto is FALSE, then in the BeginDrag event script, the programmer can call the Drag function to begin the drag operation

The user cannot drag a highlighted item.

Dragging a TreeView item onto another control causes 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. It is effective for the BeginRightDrag event too.

See also

BeginDrag
DragDrop
DragEnter
DragLeave
DragWithin

ButtonClicked

Description

Occurs when the user clicks a button.

Event ID

Event ID	Objects
pbm_dwnbuttonclicked	Button

Arguments

Argument	Description
row	Long by value (the number of the row the user clicked)
actionreturncode	Long by value (the value returned by the action performed by the button) FOR INFO For information about return values, see the Action property in the *DataWindow Reference*
dwo	DWObject by value (a reference to the object within the DataWindow under the pointer when the user clicked)

Return codes

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

0 Continue processing

Usage

Executes code after the action assigned to the button has occurred.

This event is fired only if you have not selected Suppress Event Processing for the button. If Suppress Event Processing is on, only the action assigned to the button is executed when the button is clicked.

Examples

This statement in the ButtonClicked event displays the value returned by the button's action:

```
MessageBox(" ", actionreturncode)
```

See also

ButtonClicking

ButtonClicking

Description

Occurs when the user is clicks a button. This event occurs before the ButtonClicked event.

Event ID

Event ID	Objects
pbm_dwnbuttonclicking	Button

Arguments

Argument	Description
row	Long by value (the number of the row the user clicked)
dwo	DWObject by value (a reference to the object within the DataWindow under the pointer when the user clicked)

Return codes

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

0 Execute action assigned to button followed by ButtonClicked event

Usage

Executes code before the action assigned to the button has occurred. If the return code is 0, the action assigned to the button is then executed. After the action is executed, the ButtonClicked event is fired.

This event is fired only if you have not selected Suppress Event Processing for the button.

The Clicked event is fired before the ButtonClicking event.

Examples

This statement in the ButtonClicking event displays a message box before proceeding with the action assigned to the button:

```
MessageBox(" ", "Are you sure you want to proceed?")
```

See also

ButtonClicked

Clicked

The Clicked event has different arguments for different objects:

Object	See
Menus	Syntax 1
DataWindow control	Syntax 2
ListView control	Syntax 3
Tab control	Syntax 4
TreeView control	Syntax 5
Window	Syntax 6
Other controls	Syntax 7

Syntax 1

For menus

Description

Occurs when the user chooses an item on a menu.

Event ID

Event ID	Objects
None	Menu

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:

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

See also

Selected

Syntax 2 For DataWindows

Description

Occurs when the user clicks anywhere in a DataWindow control.

Event ID

Event ID	Objects
pbm_dwnlbuttonclk	DataWindow

Arguments

Argument	Description
xpos	Integer by value (the distance of the pointer from the left side of the DataWindow's workspace)
ypos	Integer by value (the distance of the pointer from the top of the DataWindow's workspace)
row	Long by value (the number of the row the user clicked) If the user doesn't click on a row, the value of the *row* argument is 0. For example, *row* is 0 when the user clicks outside the data area, in text or spaces between rows, or in the header, summary, or footer area
dwo	DWObject by value (a reference to the object within the DataWindow under the pointer when the user clicked)

Return codes

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

0 Continue processing
1 Prevent the focus from changing

Usage

The DataWindow Clicked event occurs when the mouse button is pressed down.

The *xpos* and *ypos* arguments provide the same values the functions PointerX and PointerY return when you call them for the DataWindow control.

The *dwo* argument provides easy access to the object the user clicks. You don't need to know the coordinates of elements within the DataWindow to program object-specific responses to the user's clicks.

For example, you can prevent editing of a column and use the Clicked script to set data or properties for the column and row the user clicks.

For graphs, the ObjectAtPointer function provides similar information about objects within the graph object.

A click can also trigger RowFocusChanged and ItemFocusChanged events.

A double-click triggers a Clicked event, then a DoubleClicked event.

Examples

This code highlights the row the user clicked.

```
This.SelectRow(row, TRUE)
```

If the user clicks on a column heading, this code changes the color of the label and sorts the associated column. The column name is assumed to be the name of the heading text object without _t as a suffix.

```
string ls_name

IF dwo.Type = "text" THEN
    dwo.Color = RGB(255,0,0)

    ls_name = dwo.Name
    ls_name = Left(ls_name, Len(ls_name) - 2)

    This.SetSort(ls_name + ", A")
    This.Sort()
END IF
```

See also

DoubleClicked
ItemFocusChanged
RButtonDown
RowFocusChanged

Syntax 3 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

Event ID	Objects
pbm_lvnclicked	ListView

Arguments

Argument	Description
index	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

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.

Examples

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

```
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
ItemChanged
ItemChanging
RightClicked
RightDoubleClicked

Syntax 4

For Tab controls

Description

Occurs when the user clicks on the tab portion of a Tab control.

Event ID

Event ID	Objects
pbm_tcnclicked	Tab

Arguments

Argument	Description
index	Integer by value (the index of the tab page the user clicked)

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:

```
IF index = 4 THEN
    This.BoldSelectedText = TRUE
ELSE
    This.BoldSelectedText = FALSE
END IF
```

See also

DoubleClicked
RightClicked
RightDoubleClicked
SelectionChanged
SelectionChanging

Syntax 5 For TreeView controls

Description

Occurs when the user clicks an item in a TreeView control.

Event ID

Event ID	Objects
pbm_tvnclicked	TreeView

Arguments

Argument	Description
handle	Long by value (the handle of the TreeView item the user clicked)

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:

```
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 6 For windows

Description

Occurs when the user clicks in an unoccupied area of the window (any area with no visible, enabled object).

Event ID

Event ID	Objects
pbm_lbuttonclk	Window

Arguments

Argument	Description
flags	UnsignedLong by value (the modifier keys and mouse buttons that are pressed) Values are: ♦ 1 — Left mouse button ♦ 2 — Right mouse button ♦ 4 — SHIFT key ♦ 8 — CTRL key ♦ 16 — Middle mouse button In the Clicked event, the left mouse button is being released, so 1 is not summed in the value of *flags* FOR INFO For an explanation of *flags*, see Syntax 2 of MouseMove on page 302

Argument	Description
xpos	Integer by value (the distance of the pointer from the left edge of the window's workspace in PowerBuilder units)
ypos	Integer by value (the distance of the pointer from the top of the window's workspace in PowerBuilder units)

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 the user clicks on a control or menu, 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 the 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

DoubleClicked
MouseDown
MouseMove
MouseUp
RButtonDown

Syntax 7 For other controls

Description

Occurs when the user clicks on the control.

Event ID

Event ID	Objects
pbm_bnclicked	CheckBox, CommandButton, Graph, OLE, Picture, PictureButton, RadioButton, StaticText

Arguments

None

Return codes

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

0 Continue processing

Usage

The Clicked event occurs when the user releases the mouse button.

If another control had focus, then a GetFocus and a Clicked event occur for the control the user clicks.

Examples

This code in an OLE control's Clicked event activates the object in the control:

```
integer li_success
li_success = This.Activate(InPlace!)
```

See also

GetFocus
RButtonDown

Close

The Close event has different arguments for different objects:

Object	See
Application	Syntax 1
OLE control	Syntax 2
Window	Syntax 3

Syntax 1 For the application object

Description

Occurs when the user closes the application.

Event ID

Event ID	Objects
None	Application

Arguments

None

Return codes

None (do not use a RETURN statement)

Usage

The Close event occurs when the last window (for MDI applications the MDI frame) is closed.

See also

Open
SystemError

Syntax 2 For OLE controls

Description

Occurs when the object in an OLE control has been activated offsite (the OLE server displays the object in the server's window) and that server is closed.

Event ID

Event ID	Objects
pbm_omnclose	OLE

Arguments

None

Return codes

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

0 Continue processing

Usage

If the user closed the OLE server, the user's choices may cause the OLE object in the control to be updated, triggering the Save or DataChange events.

See also

DataChange
Save

Syntax 3 For windows

Description

Occurs just before a window is removed from display.

Event ID

Event ID	Objects
pbm_close	Window

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

Event ID	Objects
pbm_closequery	Window

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 up data and close window
    IF li_rc = 1 THEN
        Window lw_window
        lw_window = w_genapp_frame.GetActiveSheet()
        lw_window.TriggerEvent("ue_update")
        RETURN 0

    //User chose to close window without updating
    ELSEIF li_rc = 2 THEN
        RETURN 0

    //User canceled
    ELSE
        RETURN 1
    END IF

ELSE
    // No changes to the data, window will just close
    RETURN 0
END IF
```

See also

Close

ColumnClick

Description

Occurs when the user clicks a column header.

Event ID

Event ID	Objects
pbm_lvncolumnclick	ListView

Arguments

Argument	Description
column	The index of the clicked column

Return codes

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

0 Continue processing

Usage

The ColumnClicked event is only available when the ListView displays in report view and the ButtonHeader property is set to TRUE.

Examples

This example uses the ColumnClicked event to set up a instance variable for the column argument, retrieve column alignment information, and display it to the user:

```
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

ConnectionBegin

Description

In a distributed computing environment, occurs on the server when a client establishes a connection to the server by calling the ConnectToServer function.

Event ID

Event ID	Objects
None	Application

Arguments

Argument	Description
userid	The name or ID of the user who will connect to the server
password	The password that will be used to connect to the server
connectstring	Text passed to the server at connection time. The text can include application-specific information such as database connection parameters

Return codes

ConnectPrivilege enumerated data type. Return code choices (specify in a RETURN statement):

ConnectPrivilege!
ConnectWithAdminPrivilege!
NoConnectPrivilege

Usage

Whenever a client makes a request to connect to a server application, the server can validate the request. If the client has the proper authority to establish the connection, the server can permit the request and grant the appropriate connection privileges. If the client does not have the proper authority, the server can reject the request.

To handle client requests for connections, you can write a script for the ConnectionBegin event of the server's application object. If the server application accesses a database, you may also want to include the logic to connect to the database in the ConnectionBegin event.

Examples

The following script for the ConnectionBegin event validates each client connection by testing the values of *userid* and *password*, which are arguments passed to the event.

```
ConnectPrivilege access_rights

access_rights = ConnectPrivilege!
IF userid = "dba" and password = "dba" THEN
```

```
    access_rights = ConnectWithAdminPrivilege!
ELSEIF userid = "" or password = "" THEN
    access_rights = NoConnectPrivilege!
END IF
RETURN access_rights
```

See also

ConnectionEnd

ConnectionEnd

Description

In a distributed computing environment, occurs on the server when a client disconnects from the server by calling the DisconnectServer function.

Event ID

Event ID	Objects
None	Application

Arguments

Argument	Description
userid	The name or ID of the user who will be disconnected from the server
password	The password for the user who will be disconnected from the server
connectstring	Text passed to the server at connection time. The text can include application-specific information such as database connection parameters

Return codes

None (do not use a RETURN statement)

Usage

If the server application will access a database, you may want to include the logic to disconnect from the database in the ConnectionEnd event.

Examples

The following script for the ConnectionEnd event disconnects from the database:

```
DISCONNECT USING SQLCA;
```

See also

ConnectionBegin

Constructor

Description

Occurs when the control or object is created, just before the Open event for the window that contains the control.

Event ID

Event ID	Objects
pbm_constructor	All objects

Arguments

None

Return codes

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

0 Continue processing

Usage

You can write a script for a control's Constructor event to affect the control's properties before the window is displayed.

When a window or user object opens, a Constructor event for each control in the window or user object occurs. The order of controls in a window's Control property (which is an array) determines the order in which Constructor events are triggered. If one of the controls in the window is a user object, the Constructor events of all the controls in the user object occur before the Constructor event for the next control in the window.

When you call OpenUserObject to add a user object to a window dynamically, its Constructor event and the Constructor events for all of its controls occur.

When you use the CREATE statement to instantiate a class (nonvisual) user object, its Constructor event occurs.

When a class user object variable has an Autoinstantiate setting of TRUE, its Constructor event occurs when the variable comes into scope. Therefore, the Constructor event occurs for:

- Global variables when the system starts up
- Shared variables when the object with the shared variables is loaded
- Instance variables when the object with the instance variables is created
- Local variables when the function that declares them begins executing

Examples

This example retrieves data for the DataWindow dw_1 before its window is displayed:

```
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

Event ID	Objects
pbm_omndatachange	OLE

Arguments

None

Return codes

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

0 Continue processing

See also

PropertyRequestEdit
PropertyChanged
Rename
ViewChange

DBError

Description

Occurs when a database error occurs in the DataWindow or DataStore.

Event ID

Event ID	Objects
pbm_dwndberror	DataWindow or DataStore

Arguments

Argument	Description
sqldbcode	Long by value (a database-specific error code) See your DBMS documentation for information on the meaning of the code When there is no error code from the DBMS, *sqldbcode* contains one of these values: ♦ -1 — Can't connect to the database because of missing values in the transaction object ♦ -2 — Can't connect to the database ♦ -3 — The key specified in an Update or Retrieve no longer matches an existing row (this can happen when another user has changed the row after you retrieved it) ♦ -4 — Writing a blob to the database failed
sqlerrtext	String by value (a database-specific error message)
sqlsyntax	String by value (the full text of the SQL statement being sent to the DBMS when the error occurred)
buffer	DWBuffer by value (the buffer containing the row involved in the database activity that caused the error) Values are: ♦ Delete! — The delete buffer (data that has been deleted from the DataWindow) ♦ Filter! — The filter buffer (data that has been filtered out) ♦ Primary! — The primary buffer (data that has not been deleted or filtered out)
row	Long by value The number of the row involved in the database activity that caused the error (the row being updated, selected, inserted, or deleted)

Return codes

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

0 Display the error message
1 Do not display the error message

Usage

You can display your own message in the DBError event and suppress the system message with a return code of 1.

If the row that caused the error is in the Filter buffer, you must unfilter it if you want the user to correct the problem.

Reported row number
The row number stored in *row* is the number of the row in the buffer, not the number the row had when it was retrieved into the DataWindow object.

Obsolete functions Information formerly provided by the DBErrorCode and DBErrorMessage functions is available in the arguments *sqldbcode* and *sqlerrtext*.

Examples

This example illustrates how to display custom error messages for particular database error codes:

```
CHOOSE CASE sqldbcode

   CASE -195 // Required value is NULL.
      MessageBox("Database Problem", &
         "Error inserting row " + string(row) &
         + ". Please specify value for Employee Id")
   CASE ...
      // Code to handle other errors

END CHOOSE

RETURN 1 // Do not display system error message
```

See also

Error

Deactivate

Description

Occurs when the window becomes inactive.

Event ID

Event ID	Objects
pbm_deactivate	Window

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

Event ID	Objects
pbm_lvndeleteallitems	ListView

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's a default item in the ListView control:

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

See also

DeleteItem
InsertItem

DeleteItem

The DeleteItem event has different arguments for different objects:

Object	See
ListView control	Syntax 1
TreeView control	Syntax 2

Syntax 1

For ListView controls

Description

Occurs when an item is deleted.

Event ID

Event ID	Objects
pbm_lvndeleteitem	ListView

Arguments

Argument	Description
index	Integer by value (the index of the deleted item)

Return codes

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

0 Continue processing

Examples

This example for the DeleteItem event displays a message with the number of the deleted item:

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

See also

DeleteAllItems
InsertItem

Syntax 2

For TreeView controls

Description

Occurs when an item is deleted.

Event ID

Event ID	Objects
pbm_tvndeleteitem	TreeView

Arguments

Argument	Description
handle	Long by value (the handle of the deleted item)

Return codes

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

0 Continue processing

Examples

This example displays the name of the deleted item in a message:

```
TreeViewItem ll_tvi

This.GetItem(handle, ll_tvi)
MessageBox("Message", String(ll_tvi.Label) &
   + " has been deleted.")
```

Destructor

Description

Occurs when the user object or control is destroyed, immediately after the Close event of a window.

Event ID

Event ID	Objects
pbm_destructor	All objects

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've been destroyed, you can no longer refer to those controls in other scripts (if you do, a runtime error occurs).

See also

Constructor
Close

DoubleClicked

The DoubleClicked event has different arguments for different objects:

Object	See
DataWindow control	Syntax 1
ListBox, PictureListBox, ListView, and Tab controls	Syntax 2
TreeView control	Syntax 3
Window	Syntax 4
Other controls	Syntax 5

Syntax 1 For DataWindow controls

Description

Occurs when the user double-clicks in a DataWindow control.

Event ID

Event ID	Objects
pbm_dwnlbuttondblclk	DataWindow

Arguments

Argument	Description
xpos	Integer by value (the distance of the pointer from the left side of the DataWindow's workspace)
ypos	Integer by value (the distance of the pointer from the top of the DataWindow's workspace)
row	Long by value (the number of the row the user double-clicked) If the user didn't double-click on a row, the value of the *row* argument is 0. For example, *row* is 0 when the user double-clicks outside the data area, in text or spaces between rows, or in the header, summary, or footer area
dwo	DWObject by value (a reference to the object within the DataWindow the user double-clicked)

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 DataWindow control.

The *dwo* argument provides easy access to the object the user clicks. You don't need to know the coordinates of elements within the DataWindow to program object-specific responses to the user's clicks. For example, you can prevent editing of a column and use the Clicked script to set data or properties for the column and row the user clicks.

Examples

This example displays a message box reporting the row and column clicked and the position of the pointer relative to the upper-left corner of the DataWindow control:

```
string ls_columnname

IF dwo.Type = "column" THEN
   ls_columnname = dwo.Name
END IF

MessageBox("DoubleClicked Event", &
   "Row number: " + row &
   + "~rColumn name: " + ls_columnname &
   + "~rDistance from top of dw: " + ypos &
   + "~rDistance from left side of dw: " + xpos)
```

See also

Clicked
ItemFocusChanged
RButtonDown
RowFocusChanged

Syntax 2 For ListBox, PictureListBox, ListView, and Tab controls

Description

Occurs when the user double-clicks on the control.

Event ID

Event ID	Objects
pbm_lbndblclk	ListBox, PictureListBox
pbm_lvndoubleclicked	ListView
pbm_tcndoubleclicked	Tab

Arguments

Argument	Description
index	Integer by value The index of the item the user double-clicked (for tabs, the index of the tab page)

Return codes

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

0 Continue processing

Usage

In a ListView control, the Clicked event occurs twice during a double-click action, before and after the DoubleClicked event. (The Clicked event occurs the first time the button is first released; the DoubleClicked event occurs on the second click when the button is pressed; and the Clicked event occurs again when the second button press is released.)

In a ListBox or PictureListBox, double-clicking on an item also triggers a SelectionChanged event.

Examples

This example uses the DoubleClicked event to begin editing the double-clicked ListView item:

```
This.EditLabels = TRUE
```

See also

Clicked
ColumnClick
ItemChanged
ItemChanging
RightClicked
RightDoubleClicked
SelectionChanged
SelectionChanging

Syntax 3 For TreeView controls

Description

Occurs when the user double-clicks on the control.

Event ID

Event ID	Objects
pbm_tvndoubleclicked	TreeView

Arguments

Argument	Description
handle	Long by value (the handle of the item the user double-clicked)

Return codes

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

0 Continue processing

Examples

This example turns on editing for the double-clicked TreeView item:

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

See also

Clicked
RightClicked
RightDoubleClicked
SelectionChanged
SelectionChanging

Syntax 4 For windows

Description

Occurs when the user double-clicks in an unoccupied area of the window (any area with no visible, enabled object).

Event ID

Event ID	Objects
pbm_lbuttondblclk	Window

Arguments

Argument	Description
flags	UnsignedLong by value (the modifier keys and mouse buttons that are pressed) Values are: ♦ 1 — Left mouse button ♦ 2 — Right mouse button ♦ 4 — SHIFT key ♦ 8 — CTRL key ♦ 16 — Middle mouse button In the Clicked event, the left mouse button is being released, so 1 is not summed in the value of *flags* FOR INFO For an explanation of *flags*, see Syntax 2 of MouseMove on page 302
xpos	Integer by value (the distance of the pointer from the left edge of the window's workspace in PowerBuilder units)

Argument	Description
ypos	Integer by value (the distance of the pointer from the top of the window's workspace in PowerBuilder units)

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

Syntax 5 For other controls

Description

Occurs when the user double-clicks on the control.

Event ID

Event ID	Objects
pbm_bndoubleclicked	Graph, OLE, Picture, StaticText
pbm_cbndblclk	DropDownListBox, DropDownPictureListBox
pbm_rendoubleclicked	RichTextEdit

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.

On Macintosh
Since this property is not available on the Macintosh, the DoubleClicked event is not available for DropDownListBoxes on the Macintosh.

See also

Clicked
RButtonDown

DragDrop

The DragDrop event has different arguments for different objects:

Object	See
DataWindow control	Syntax 1
ListBox, PictureListBox, ListView, and Tab controls	Syntax 2
TreeView control	Syntax 3
Windows and other controls	Syntax 4

Syntax 1

For DataWindow controls

Description

Occurs when the user drags an object onto the control and releases the mouse button to drop the object.

Event ID

Event ID	Objects
pbm_dwndragdrop	DataWindow

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)
row	Long by value (the number of the row the pointer was over when the user dropped the object) If the pointer wasn't over a row, the value of the *row* argument is 0. For example, *row* is 0 when the pointer is outside the data area, in text or spaces between rows, or in the header, summary, or footer area
dwo	DWObject by value (a reference to the object under the pointer within the DataWindow when the user dropped the object)

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 the DragDrop event for a DataWindow checks whether the source object is a DataWindow control. If so, it finds out the current row in the source and moves it to the target:

```
DataWindow ldw_Source

IF source.TypeOf() = DataWindow! THEN
   ldw_Source = source
   IF row > 0 THEN
      ldw_Source.RowsMove(row, row, Primary!, &
         This, 1, Primary!)
   END IF
END IF
```

See also

DragEnter
DragLeave
DragWithin

Syntax 2 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

Event ID	Objects
pbm_lbndragdrop	ListBox, PictureListBox
pbm_lvndragdrop	ListView
pbm_tcndragdrop	Tab

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)
index	Integer by value (the index of the target ListView item)

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:

```
This.AddItem(ilvi_dragged_object)
This.Arrange( )
```

See also

BeginDrag
BeginRightDrag
DragEnter
DragLeave
DragWithin

Syntax 3 — 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

Event ID	Objects
pbm_tvndragdrop	TreeView

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)
handle	Long by value (the handle of the target TreeView item)

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:

```
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 4

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

Event ID	Objects
pbm_bndragdrop	CheckBox, CommandButton, Graph, Picture, PictureButton, RadioButton
pbm_cbndragdrop	DropDownListBox, DropDownPictureListBox
pbm_endragdrop	SingleLineEdit, EditMask, MultiLineEdit, StaticText
pbm_omndragdrop	OLE
pbm_rendragdrop	RichTextEdit
pbm_sbndragdrop	HScrollBar, VScrollBar
pbm_uondragdrop	UserObject
pbm_dragdrop	Window

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)

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 from the w_orderentry window in the ABNC sample application, this code in the DoubleClicked event for the DataWindow dw_orddetail starts a drag operation:

```
IF dw_orddetail.GetRow() > 0 THEN
   dw_orddetail.Drag(Begin!)
   This.DragIcon = "dragitem.ico"
```

```
END IF
```

Then in the DragDrop event for a trashcan Picture control, this code deletes the row the user clicked and dragged from the DataWindow control:

```
long ll_currow
dwitemstatus ldwis_delrow

ll_currow = dw_orddetail.GetRow( )

// Save the row's status flag for later use
ldwis_delrow = dw_orddetail.GetItemStatus &
   (ll_currow, 0, Primary!)

// Now, delete the current row from dw_orddetail
dw_orddetail.DeleteRow(0)
```

Example 2 This example for a trashcan Picture control's DragDrop event checks whether the source of the drag operation is a DataWindow. If so, it asks the user whether to delete the current row in the source DataWindow:

```
DataWindow ldw_Source
Long ll_RowToDelete
Integer li_Choice

IF source.TypeOf() = DataWindow! THEN

   ldw_Source = source
   ll_RowToDelete = ldw_Source.GetRow()

   IF ll_RowToDelete > 0 THEN
      li_Choice = MessageBox("Delete", &
         "Delete this row?", Question!, YesNo!, 2)
      IF li_Choice = 1 THEN
         ldw_Source.DeleteRow(ll_RowToDelete)
      END IF
   ELSE
      Beep(1)
   END IF

ELSE
   Beep(1)
END IF
```

See also

DragEnter
DragLeave
DragWithin

DragEnter

Description

Occurs when the user is dragging an object and enters the control.

Event ID

Event ID	Objects
pbm_bndragenter	CheckBox, CommandButton, Graph, Picture, PictureButton, RadioButton
pbm_cbndragenter	DropDownListBox, DropDownPictureListBox
pbm_dwndragenter	DataWindow
pbm_endragenter	SingleLineEdit, EditMask, MultiLineEdit, StaticText
pbm_lbndragenter	ListBox, PictureListBox
pbm_lvndragenter	ListView
pbm_omndragenter	OLE
pbm_rendragenter	RichTextEdit
pbm_sbndragenter	HScrollBar, VScrollBar
pbm_tcndragenter	Tab
pbm_tvndragenter	TreeView
pbm_uondragenter	UserObject
pbm_dragenter	Window

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)

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

Event ID	Objects
pbm_bndragleave	CheckBox, CommandButton, Graph, Picture, PictureButton, RadioButton
pbm_cbndragleave	DropDownListBox, DropDownPictureListBox
pbm_dwndragleave	DataWindow
pbm_endragleave	SingleLineEdit, EditMask, MultiLineEdit, StaticText
pbm_lbndragleave	ListBox, PictureListBox
pbm_lvndragleave	ListView
pbm_omndragleave	OLE
pbm_rendragleave	RichTextEdit
pbm_sbndragleave	HScrollBar, VScrollBar
pbm_tcndragleave	Tab
pbm_tvndragleave	TreeView
pbm_uondragleave	UserObject
pbm_dragleave	Window

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)

Return codes

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

0 Continue processing

Usage

Obsolete functions You no longer need to call the DraggedObject function in a drag event. Use the *source* argument instead.

Examples

This example checks the name of the control being dragged and if it is cb_1 it cancels the drag operation:

```
IF ClassName(source) = "cb_1" THEN
   cb_1.Drag(Cancel!)
END If
```

This example for a Picture control's DragDrop event removes its own border when another Picture control (the source) is dragged beyond its boundaries:

```
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:

Object	See
DataWindow control	Syntax 1
ListBox, PictureListBox, ListView, and Tab controls	Syntax 2
TreeView control	Syntax 3
Windows and other controls	Syntax 4

Syntax 1 For DataWindow controls

Description

Occurs when the user is dragging an object within the control.

Event ID

Event ID	Objects
pbm_dwndragwithin	DataWindow

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)
row	Long by value (the number of the row the pointer is over) If the pointer isn't over a row, the value of the *row* argument is 0. For example, *row* is 0 when the pointer is outside the data area, in text or spaces between rows, or in the header, summary, or footer area
dwo	DWObject by value (a reference to the object under the pointer within the DataWindow)

Return codes

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

0 Continue processing

Usage

The DragWithin event occurs repeatedly as the mouse moves within the control.

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

Syntax 2

For ListBox, PictureListBox, ListView, and Tab controls

Description

Occurs when the user is dragging an object within the control.

Event ID

Event ID	Objects
pbm_lbndragwithin	ListBox, PictureListBox
pbm_lvndragwithin	ListView
pbm_tcndragwithin	Tab

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)
index	Integer by value (a reference to the ListView item under the pointer in the ListView control)

Return codes

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

0 Continue processing

Usage

Obsolete functions You no longer need to call the DraggedObject function in a drag event. Use the *source* argument instead.

Examples

This example changes the background color of the ListView when a DragObject enters its border:

```
This.BackColor = RGB(128, 0, 128)
```

See also

DragDrop
DragEnter
DragLeave

Syntax 3

For TreeView controls

Description

Occurs when the user is dragging an object within the control.

Event ID

Event ID	Objects
pbm_tvndragwithin	TreeView

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)
handle	Long (a reference to the ListView item under the pointer in the TreeView control)

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 4 For windows and other controls

Description

Occurs when the user is dragging an object within the control.

Event ID

Event ID	Objects
pbm_bndragwithin	CheckBox, CommandButton, Graph, Picture, PictureButton, RadioButton
pbm_cbndragwithin	DropDownListBox, DropDownPictureListBox
pbm_endragwithin	SingleLineEdit, EditMask, MultiLineEdit, StaticText
pbm_omndragwithin	OLE
pbm_rendragwithin	RichTextEdit
pbm_sbndragwithin	HScrollBar, VScrollBar
pbm_uondragwithin	UserObject

Event ID	Objects
pbm_dragwithin	Window

Arguments

Argument	Description
source	DragObject by value (a reference to the control being dragged)

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

EditChanged

Description

Occurs for each keystroke the user types in an edit control in the DataWindow.

Event ID

Event ID	Objects
pbm_dwnchanging	DataWindow

Arguments

Argument	Description
row	Long by value (the number of the row containing the item whose value is being changed)
dwo	DWObject by value (a reference to the column containing the item whose value is being changed. *Dwo* is a reference to the column object, not the name of the column)
data	String by value (the current contents of the DataWindow edit control)

Return codes

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

0 Continue processing

Examples

This example displays the row and column that the user is editing in a StaticText control:

```
st_1.Text = "Row " + String(row) &
   + " in column " + dwo.Name
```

See also

ItemChanged

EndLabelEdit

The EndLabelEdit event has different arguments for different objects:

Object	See
ListView control	Syntax 1
TreeView control	Syntax 2

Syntax 1 For ListView controls

Description

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

Event ID

Event ID	Objects
pbm_lvnendlabeledit	ListView

Arguments

Argument	Description
index	Integer. The index of the ListView item for which you have edited the label
newlabel	The string that represents the new label for the ListView item

Return codes

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

0 Allow the new text to become the item's label
1 Prevent the new text from becoming the item's label

Usage

The user triggers this event by pressing ENTER or TAB after editing the text.

Examples

This example displays the old label and the new label in a SingleLineEdit:

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

See also

BeginLabelEdit

Syntax 2 For TreeView controls

Description

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

Event ID

Event ID	Objects
pbm_tvnendlabeledit	TreeView

Arguments

Argument	Description
handle	Integer. The index of the TreeView item for which you have edited the label
newtext	The string that represents the new label for the TreeView item

Return codes

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

0 Allow the new text to become the item's label
1 Prevent the new text from becoming the item's label

Usage

The user triggers this event by pressing ENTER or TAB after editing the text.

Examples

This example displays the old label and the new label in a SingleLineEdit:

```
TreeViewItem tvi

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

See also

BeginLabelEdit

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 distributed application.

Event ID

Event ID	Objects
None	Connection, DataWindow, DataStore, OLE, OLEObject

Arguments

Argument	Description
errornumber	Unsigned integer by value (PowerBuilder's error number)
errortext	String, read-only (PowerBuilder's error message)
errorwindowmenu	String, read-only (the name of the window or menu that is the parent of the object whose script caused the error)
errorobject	String, read-only (the name of the object whose script caused the error)
errorscript	String, read-only (the full text of the script in which the error occurred)
errorline	Unsigned integer by value (the line in the script where the error occurred)
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. This option is not valid for DataWindows ♦ 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

Argument	Description
returnvalue	Any by reference (a value whose data type matches the expected value that the OLE server or DataWindow would have returned) This value is used when the value of *action* is ExceptionSubstituteReturnValue!

Return codes

None (do not use a RETURN statement)

Usage

DataWindow and OLE objects are dynamic. Expressions that use dot notation to refer to data and properties of these objects may 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 can't be checked by the compiler—such expressions can only be evaluated during execution.

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 ExceptionSubstituteReturnValue! to substitute a return value when an element in the middle of an expression causes an error. The substituted return value will not match the data type of the unresolved object reference and will cause a system error.

The ExceptionSubstituteReturnValue! action is most 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!, the SystemError event occurs
3. 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*.

For information about handling communications errors in a distributed application, see the discussion of distributed applications in *Application Techniques*.

Examples

This example displays information about the error that occurred and allows the script to continue:

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

See also

DBError
ExternalException
SystemError

ExternalException

Description

Occurs when an OLE automation command caused an exception on the OLE server.

Event ID

Event ID	**Objects**
None	OLE, OLEObject

Arguments

Argument	**Description**
resultcode	UnsignedLong by value (a PowerBuilder number identifying the exception that occurred on the server)
exceptioncode	UnsignedLong by value (a number identifying the error that occurred on the server. For the meaning of the code, see the server documentation)
source	String by value (the name of the server (provided by the server))
description	String by value (a description of the exception (provided by the server))
helpfile	String by value (the name of a Help file containing information about the exception (provided by the server))
helpcontext	UnsignedLong by value (the context ID of a Help topic in *helpfile* containing information about the exception (provided by the server))

Argument	Description
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 data type 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 may 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 data types, generally do not cause an exception but do trigger the Error event. In some cases you may 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!, the SystemError event occurs
4. 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 an appropriate values:

```
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. The assignment of the substitute value to li_currentsetting works correctly because their data types are compatible:

```
string ls_context

// Command line switch for WinHelp numeric context ID
ls_context = "-n " + String(helpcontext)
If Len(HelpFile) > 0 THEN
   Run("winhelp.ext " + ls_context + " " + helpfile)
END IF

action = ExceptionSubstituteReturnValue!
returnvalue = ia_substitute
```

Because the event script must serve for every automation command for the control, you would 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

Event ID	Objects
pbm_renfileexists	RichTextEdit

Arguments

Argument	Description
filename	The name of the file

Return codes

Long. Return code choices (specify 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,
// don't 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
```

GetFocus

Description

Occurs just before the control receives focus (before it is selected and becomes active).

Applies to all controls

Event ID

Event ID	Objects
pbm_bnsetfocus	CheckBox, CommandButton, Graph, OLE, Picture, PictureButton, RadioButton
pbm_cbnsetfocus	DropDownListBox, DropDownPictureListBox
pbm_dwnsetfocus	DataWindow
pbm_ensetfocus	SingleLineEdit, EditMask, MultiLineEdit, StaticText
pbm_lbnsetfocus	ListBox, PictureListBox
pbm_lvnsetfocus	ListView
pbm_rensetfocus	RichTextEdit
pbm_sbnsetfocus	HScrollBar, VScrollBar
pbm_tcnsetfocus	Tab
pbm_tvnsetfocus	TreeView

Arguments

None

Return codes

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

0 Continue processing

Usage

On Macintosh

On the Macintosh, only controls that have editable text can have focus. Other controls never get focus. A GetFocus event never occurs for controls without an edit box. So if you are designing an application that you want to run on the Macintosh as well as other PowerBuilder platforms, you should not put important code in GetFocus and LoseFocus event scripts.

For example if a user edits text in a SingleLineEdit and then clicks on a CommandButton, the SingleLineEdit still has the focus. The Clicked event occurs for the CommandButton, but no LoseFocus or GetFocus events occur.

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

Hide

Description

Occurs just before the window is hidden.

Event ID

Event ID	Objects
pbm_hidewindow	Window

Arguments

None

Return codes

Long. Return code choices (specify 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 popup 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

Event ID	Objects
pbm_ddedata	Window

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:

```
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

Event ID	Objects
None	Application

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 has double-clicked or pressed ENTER in an input field, allowing the user to edit the data in the field.

Event ID

Event ID	Objects
pbm_reninputfieldselected	RichTextEdit

Arguments

Argument	Description
fieldname	String by value (the name of the input field that was selected)

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

Event ID	Objects
pbm_lvninsertitem	ListView

Arguments

Argument	Description
index	An integer that represents the index of the item being inserted into the TreeView

Return codes

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

0 Continue processing

Examples

This example displays the label and index of the inserted item:

```
ListViewItem lvi
This.GetItem(index, lvi)
sle_info.Text = "Inserted "+ String(lvi.Label) &
    + " into position " &
    + String(index)
```

See also

DeleteItem

ItemChanged

The ItemChanged event has different arguments for different objects:

Object	See
DataWindow control and DataStore	Syntax 1
ListView control	Syntax 2

Syntax 1

For DataWindow controls and DataStores

Description

Occurs when a field in a DataWindow control has been modified and loses focus (for example, the user presses ENTER, the TAB key, or an arrow key or clicks the mouse on another field within the DataWindow). ItemChanged can also occur when the AcceptText or Update function is called for a DataWindow control or DataStore object.

Event ID

Event ID	Objects
pbm_dwnitemchange	DataWindow control or DataStore

Arguments

Argument	Description
row	Long by value (the number of the row containing the item whose value has been changed)
dwo	DWObject by value (a reference to the column containing the item whose value has been changed. *Dwo* is a reference to the column object, not the name of the column)
data	String by value (the new data the user has specified for the item)

Return codes

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

0 (Default) Accept the data value
1 Reject the data value and don't allow focus to change
2 Reject the data value but allow the focus to change

Usage

The ItemChanged event does not occur when the DataWindow control itself loses focus. If the user clicks on an Update or Close button, you will need to write a script that calls AcceptText to see if a changed value should be accepted before the button's action occurs.

Obsolete techniques
Information formerly provided by the GetText function is available in the *data* argument.

Instead of calling SetActionCode, use a RETURN statement with a return code.

Examples

This example uses the ItemChanged event to provide additional validation; if the column is emp_name, it checks that only letters were entered in the column:

```
IF dwo.Name = "emp_name" THEN
    IF NOT Match(data, ^[A-Za-z]$) THEN
       RETURN 2
    END IF
END IF
```

See also

ItemError

Syntax 2 For ListView controls

Description

Occurs when an ListView item has changed.

Event ID

Event ID	Objects
pbm_lvnitemchanged	ListView

Arguments

Argument	Description
index	The index of the item that is changing
focuschanged	Boolean (specifies if focus has changed for the item)
hasfocus	Boolean (specifies whether the item has focus)
selectionchange	Boolean (specifies whether the selection has changed for the item)
selected	Boolean (specifies whether the item is selected)
otherchange	Boolean (specifies if anything other than focus or selection has changed for the item)

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:

```
ListViewItem l_lvi
lv_list.GetItem(index, l_lvi)
IF focuschange and hasfocus THEN
sle1.Text = String(lvi.label) +" has gotten focus."
```

See also

ItemChanging

ItemChanging

Description

Occurs just before a ListView changes.

Event ID

Event ID	Objects
pbm_lvnitemchanging	ListView

Arguments

Argument	Description
index	The index of the item that has changed
focuschange	Boolean (specifies if focus is changing for the item)
hasfocus	Boolean (specifies whether the item has focus)
selectionchange	Boolean (specifies whether the selection is changing for the item)
selected	Boolean (specifies whether the item is selected)
otherchange	Boolean (specifies if anything other than focus or selection has changed for the item)

Return codes

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

0 Continue processing

See also

ItemChanged

ItemCollapsed

Description

Occurs when a TreeView item has collapsed.

Event ID

Event ID	Objects
pbm_tvnitemcollapsed	TreeView

Arguments

Argument	Description
handle	Long by reference (the handle of the collapsed TreeViewItem)

Return codes

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

0 Continue processing

Examples

This example changes the picture for the collapsed item:

```
TreeViewItem l_tvi
integer li_level

This.GetItem(handle, l_tvi)

CHOOSE CASE l_tvi.Level
    CASE 1
        l_tvi.PictureIndex = 1
        l_tvi.SelectedPictureIndex = 1
    CASE 2
        l_tvi.PictureIndex = 2
        l_tvi.SelectedPictureIndex = 2
    CASE 3
        l_tvi.PictureIndex = 3
        l_tvi.SelectedPictureIndex = 3
    CASE 4
        l_tvi.PictureIndex = 4
        l_tvi.SelectedPictureIndex = 4
END CHOOSE
This.SetItem(handle, l_tvi)
```

See also

ItemCollapsing

ItemCollapsing

Description

Occurs when a TreeView item is collapsing.

Event ID

Event ID	Objects
pbm_tvnitemcollapsing	TreeView

Arguments

Argument	Description
handle	Long by reference (the handle of the collapsing item)

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:

```
TreeViewItem l_tvi
integer li_level

This.GetItem(handle, l_vti)

CHOOSE CASE l_tvi.level
    CASE 1
        l_tvi.PictureIndex = 1
        l_tvi.SelectedPictureIndex = 1
    CASE 2
        l_tvi.PictureIndex = 2
        l_tvi.SelectedPictureIndex = 2
    CASE 3
        l_tvi.PictureIndex = 3
        l_tvi.SelectedPictureIndex = 3
    CASE 4
        l_tvi.PictureIndex = 4
        l_tvi.SelectedPictureIndex = 4
END CHOOSE

This.SetItem(handle, l_tvi)
```

See also

ItemCollapsed

ItemError

Description

Occurs when a field has been modified, the field loses focus (for example, the user presses ENTER, TAB, or an arrow key or clicks the mouse on another field in the DataWindow), and the data in the field does not pass the validation rules for its column. ItemError can also occur when a value imported into a DataWindow control or DataStore does not pass the validation rules for its column.

Event ID

Event ID	Objects
pbm_dwnitemvalidationerror	DataWindow control or DataStore

Arguments

Argument	Description
row	Long by value (the number of the row containing the item whose value has been changed and has failed validation)
dwo	DWObject by value (a reference to the column containing the item whose value has failed validation. *Dwo* is a reference to the column object, not the name of the column)
data	String by value (the new data the user specified for the item)

Return codes

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

0 (Default) Reject the data value and show an error message box
1 Reject the data value with no message box
2 Accept the data value
3 Reject the data value but allow focus to change

Usage

If the Return code is 0 or 1 (rejecting the data) the field with the incorrect data regains the focus.

The ItemError event occurs instead of the ItemChanged event when the new data value fails a validation rule. You can force the ItemError event to occur by rejecting the value in the ItemChanged event.

Obsolete techniques
Information provided by the GetText and GetRow functions is now available in the *data* and *row* arguments.

Instead of calling GetColumnName, use the *dwo* argument and a reference to its Name property.

Instead of calling SetActionCode, use a RETURN statement with the return codes listed above.

Examples

The following excerpt from an ItemError event script of a DataWindow control allows the user to blank out a column and move to the next column. For columns with data types other than string, the user cannot leave the value empty (the empty string doesn't match the data type). If the user tried to leave the value blank, this code sets the value of the column to a NULL value of the appropriate data type.

```
string ls_colname, ls_datatype

ls_colname = dwo.Name
ls_datatype = dwo.ColType

// Reject the value if non-blank
IF Trim(data) <> "" THEN
   RETURN 0
END IF

// Set value to null if blank
CHOOSE CASE ls_datatype

   CASE "long"
      integer null_num
      SetNull(null_num)
      This.SetItem(row, ls_colname, null_num)
      RETURN 3

   CASE "date"
      date null_date
      SetNull(null_date)
      This.SetItem(row, ls_colname, null_date)
      RETURN 3
```

```
    // Additional cases for other data types
END CHOOSE
```

See also

ItemChanged

ItemExpanded

Description

Occurs when a TreeView item has expanded.

Event ID

Event ID	Objects
pbm_tvnitemexpanded	TreeView

Arguments

Argument	Description
handle	Long by reference (the handle of the expanded item)

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:

```
TreeViewItem l_tvi
integer li_level

This.GetItem(handle, l_tvi)

CHOOSE CASE l_tvi.Level
    CASE 1
        l_tvi.PictureIndex = 5
        l_tvi.SelectedPictureIndex = 1
    CASE 2
        l_tvi.PictureIndex = 5
        l_tvi.SelectedPictureIndex = 2
    CASE 3
        l_tvi.PictureIndex = 5
        l_tvi.SelectedPictureIndex = 3
    CASE 4
        l_tvi.PictureIndex = 4
        l_tvi.SelectedPictureIndex = 5
END CHOOSE

This.SetItem(handle, l_tvi)
```

See also

ItemExpanding

ItemExpanding

Description

Occurs while a TreeView item is expanding.

Event ID

Event ID	Objects
pbm_tvnitemexpanding	TreeView

Arguments

Argument	Description
handle	Long by reference (the handle of the expanding TreeView item)

Return codes

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

0 Continue processing

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
    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

ItemFocusChanged

Description

Occurs when the current item in the control changes.

Event ID

Event ID	Objects
pbm_dwnitemchangefocus	DataWindow control

Arguments

Argument	Description
row	Long by value (the number of the row containing the item that just gained focus)
dwo	DWObject by value (a reference to the column containing the item that just gained focus)

Return codes

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

0 Continue processing

Usage

ItemFocusChanged occurs when focus is set to another column in the DataWindow, including when the DataWindow is first displayed.

In the ItemFocusChanged event, *dwo* is always a column object. Therefore, you can get more information about it by examining any properties that are appropriate for columns such as dwo.id and dwo.Name.

The row and column together uniquely identify an item in the DataWindow.

Examples

This example reports the row and column that has just gained focus and that just lost focus (the first time the event occurs, there is no item that just lost focus; the script saves the row number and column name in two instance variables called ii_row and is_colname so that the old item is known the next time the event occurs):

```
IF ii_row > 0 THEN
   sle_olditem.Text = "Old row: " + String(ii_row) &
   + " Old column: " + is_colname
END IF

sle_newitem.Text = "New row: " + String(row) &
   + " New column: " + dwo.Name

// Replace values of instance variables
// with info for next change in focus
ii_row = row
```

```
is_colname = dwo.Name
```

See also

RowFocusChanged

ItemPopulate

Description

Occurs when a TreeView item is being populated with children.

Event ID

Event ID	Objects
pbm_tvnitempopulate	TreeView

Arguments

Argument	Description
handle	Long by reference (the handle of the TreeView item being populated)

Return codes

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

0 Continue processing

Examples

This example displays the name of the TreeView item you are populating in a SingleLineEdit:

```
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

Event ID	Objects
pbm_lvnkeydown	ListView
pbm_renkey	RichTextEdit
pbm_tcnkeydown	Tab
pbm_tvnkeydown	TreeView
pbm_keydown	Window

Arguments

Argument	Description
key	KeyCode by value A value of the KeyCode enumerated data type indicating the key that was pressed (for example, KeyA! or KeyF1!)
keyflags	UnsignedLong by value (the modifier keys that were pressed with the key) Values are: 1 SHIFT key 2 CTRL key 3 SHIFT and CTRL keys

Return codes

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

0 Continue processing
1 Do not process the key (RichTextEdit controls only)

Usage

Some PowerBuilder controls capture keystrokes so that the window is prevented from getting a Key event. These include ListView, TreeView, Tab, RichTextEdit, and the DataWindow edit control. When these controls have focus you can respond to keystrokes by writing a script for an event for the control. If there is no predefined event for keystrokes, you can define a user event and associate it with a pbm code.

For a RichTextEdit control, pressing a key can perform document formatting. For example, CTRL+B applies bold formatting to the selection. If you specify a return value of 1, the document formatting associated with the key will not be performed.

If the user presses a modifier key and holds it down while pressing another key, the Key event occurs twice: once when the modifier key is pressed and again when the second key is pressed. If the user releases the modifier key before pressing the second key, the value of *keyflags* will change in the second occurrence.

When the user releases a key, the Key event does not occur. Therefore, if the user releases a modifier key, you don't 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:

```
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.

```
st_1.Text = String(keyflags)
```

See also

SystemKey

LineDown

Description

Occurs when the user clicks the down arrow of the vertical scrollbar.

Event ID

Event ID	Objects
pbm_sbnlinedown	VScrollBar

Arguments

None

Return codes

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

0 Continue processing

Usage

When the user clicks in a vertical scrollbar, nothing happens unless you have scripts that change the scrollbar's Position property. For the scrollbar arrows, use the LineUp and LineDown events; for clicks in the scrollbar 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 scrollbar and displays the resulting position in the StaticText control st_1:

```
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 scrollbar.

Event ID

Event ID	Objects
pbm_sbnlineup	HScrollBar

Arguments

None

Return codes

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

0 Continue processing

Usage

When the user clicks in a horizontal scrollbar, nothing happens unless you have scripts that change the scrollbar's Position property. For the scrollbar arrows, use the LineLeft and LineRight events; for clicks in the scrollbar 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 scrollbar 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 = "LineLeft " + String(This.Position)
```

See also

LineDown
LineRight
LineUp
PageLeft

LineRight

Description

Occurs when right arrow of the horizontal scrollbar is clicked.

Event ID

Event ID	Objects
pbm_sbnlinedown	HScrollBar

Arguments

None

Return codes

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

0 Continue processing

Usage

When the user clicks in a horizontal scrollbar, nothing happens unless you have scripts that change the scrollbar's Position property. For the scrollbar arrows, use the LineLeft and LineRight events; for clicks in the scrollbar background above and below the thumb, use the PageLeft and PageRight 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 scrollbar and displays the resulting position in the StaticText control st_1:

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

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

See also

LineDown
LineLeft
LineUp
PageRight

LineUp

Description

Occurs when the up arrow of the vertical scrollbar is clicked.

Event ID

Event ID	Objects
pbm_sbnlineup	VScrollBar

Arguments

None

Return codes

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

0 Continue processing

Usage

When the user clicks in a vertical scrollbar, nothing happens unless you have scripts that change the scrollbar's Position property. For the scrollbar arrows, use the LineUp and LineDown events; for clicks in the scrollbar background above and below the thumb, use the PageUp and PageDown events; 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 scrollbar 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 receives focus (before it becomes selected and active).

Event ID

Event ID	Description
pbm_bnkillfocus	UserObject, standard visual user objects only
pbm_bnkillfocus	CheckBox, CommandButton, Graph, OLE, Picture, PictureButton, RadioButton, StaticText
pbm_cbnkillfocus	DropDownListBox, DropDownPictureListBox
pbm_dwnkillfocus	DataWindow
pbm_enkillfocus	SingleLineEdit, EditMask, MultiLineEdit
pbm_lbnkillfocus	ListBox, PictureListBox
pbm_lvnkillfocus	ListView
pbm_renkillfocus	RichTextEdit
pbm_sbnkillfocus	HScrollBar, VScrollBar
pbm_tcnkillfocus	Tab
pbm_tvnkillfocus	TreeView

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.

On Macintosh
Only editable controls can have focus; so a LoseFocus event does *not* occur when the user clicks on a button or other noneditable control.

Examples

Example 1 In this script for the LoseFocus event of a SingleLineEdit sle_town, the user is reminded to enter information if the textbox 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()
IF TypeOf(l_control) = RichTextEdit! THEN RETURN 0

// Perform processing only if RichTextEdit lost focus
...
```

This script gets the name of the control instead:

```
GraphicObject l_control
string ls_name
l_control = GetFocus()
ls_name = l_control.Classname( )
```

See also

GetFocus

Modified

Description

Occurs when the contents in the control has changed.

Event ID

Event ID	Objects
pbm_cbnmodified	DropDownListBox, DropDownPictureListBox
pbm_enmodified	SingleLineEdit, EditMask, MultiLineEdit
pbm_renmodified	RichTextEdit

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 RichText Edit controls, the value of the Modified property controls the Modified event. If the property is FALSE, the event occurs when the first change occurs to the contents of the control. The change also causes the property to be set to TRUE, which suppresses the Modified event. You can restart checking for changes by setting the property back to FALSE.

Resetting the Modified property is useful when you insert 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 didn't enter RED, WHITE, or BLUE, a message box tells them what is valid input; for valid input, the color of the text changes):

```
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 listbox; in a real situation, the allowed input might be more general.

See also

LoseFocus

MouseDown

The MouseDown event has different arguments for different objects:

Object	See
RichTextEdit control	Syntax 1
Window	Syntax 2

Syntax 1

For RichTextEdit controls

Description

Occurs when the user presses the left mouse button on the RichTextEdit control.

Event ID

Event ID	Objects
pbm_renlbuttondown	RichTextEdit

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

Event ID	Objects
pbm_lbuttondown	Window

Arguments

Argument	Description
flags	UnsignedLong by value (the modifier keys and mouse buttons that are pressed) Values are: ♦ 1 — Left mouse button ♦ 2 — Right mouse button ♦ 4 — SHIFT key ♦ 8 — CTRL key ♦ 16 — Middle mouse button In the MouseDown event, the left mouse button is always down, so 1 is always summed in the value of *flags* FOR INFO For an explanation of *flags*, see Syntax 2 of MouseMove on page 302
xpos	Integer by value (the distance of the pointer from the left edge of the window's workspace in PowerBuilder units)
ypos	Integer by value (the distance of the pointer from the top of the window's workspace in PowerBuilder units)

Return codes

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

0 Continue processing

Examples

Example 1 This code in the MouseDown event displays the window coordinates of the pointer as reported in the *xpos* and *ypos* arguments:

```
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_modkey:

```
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:

Object	See
RichTextEdit control	Syntax 1
Window	Syntax 2

Syntax 1

For RichTextEdit controls

Description

Occurs when the mouse has moved within the RichTextEdit control.

Event ID

Event ID	Objects
pbm_renmousemove	RichTextEdit

Arguments

None

Return codes

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

0 Continue processing

See also

Clicked
MouseDown
MouseUp

Syntax 2

For windows

Description

Occurs when the pointer is moved within the window.

Event ID

Event ID	Objects
pbm_mousemove	Window

Arguments

Argument	Description
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 PowerBuilder units)
ypos	Integer by value (the distance of the pointer from the top of the window's workspace in PowerBuilder units)

Return codes

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

0 Continue processing

Usage

Because most controls in the window do not capture MouseMove events, the window's MouseMove event will still be triggered when the mouse moves over the controls.

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):

```
boolean lb_left_button, lb_right_button
boolean lb_middle_button, lb_shift_key, lb_control_key
integer li_flags

li_flags = flags
IF li_flags > 15 THEN
   // Middle button is pressed
```

```
    lb_middle_button = TRUE
    li_flags = li_flags - 16
END IF

IF li_flags > 7 THEN
    // Control key is pressed
    lb_control_key = TRUE
    li_flags = li_flags - 8
END IF

IF li_flags > 3 THEN
    // Shift key is pressed
    lb_shift_key = TRUE
    li_flags = li_flags - 4
END IF

IF li_flags > 1 THEN
    // Right button is pressed
    lb_lb_right_button = TRUE
    li_flags = li_flags - 2
END IF

IF li_flags = 1 THEN lb_left_button = TRUE
```

Examples

This code in the MouseMove event causes a meter OLE custom control (OCX) 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() )
```

The uf_setmonitor is a window function that scales the PowerBuilder units 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
```

Because the OCX also has a MouseMove event, this code in its MouseMove event keeps the gauge responding when the pointer is over the gauge (you need to pass values for the arguments the system ordinarily handles; the mouse position values are specified in relation to the parent window):

```
Parent.EVENT MouseMove(0, Parent.PointerX(), &
Parent.PointerY())
```

See also

Clicked
MouseDown
MouseUp

MouseUp

The MouseUp event has different arguments for different objects:

Object	See
RichTextEdit control	Syntax 1
Window	Syntax 2

Syntax 1 — For RichTextEdit controls

Description

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

Event ID

Event ID	Objects
pbm_renlbuttonup	RichTextEdit

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

Event ID	Objects
pbm_lbuttonup	Window

Arguments

Argument	Description
flags	UnsignedLong by value (the modifier keys and mouse buttons that are pressed) Values are: ♦ 1 — Left mouse button ♦ 2 — Right mouse button ♦ 4 — SHIFT key ♦ 8 — CTRL key ♦ 16 — Middle mouse button In the MouseUp event, the left mouse button is being released, so 1 is not summed in the value of *flags* FOR INFO For an explanation of *flags*, see Syntax 2 of MouseMove on page 302
xpos	Integer by value (the distance of the pointer from the left edge of the window's workspace in PowerBuilder units)
ypos	Integer by value (the distance of the pointer from the top of the window's workspace in PowerBuilder units)

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.

```
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.

```
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

Event ID	Objects
pbm_sbnthumbtrack	HScrollBar, VScrollBar

Arguments

Argument	Description
scrollpos	Integer by value (a number indicating position of the scroll box within the range of values specified by the MinPosition and MaxPosition properties)

Return codes

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

0 Continue processing

Usage

The Moved event updates the Position property of the scrollbar with the value of *scrollpos*.

Examples

This statement in the Moved event displays the new position of the scroll box in a StaticText control:

```
st_1.Text = "Moved " + String(scrollpos)
```

See also

LineDown
LineLeft
LineRight
LineUp
PageDown
PageLeft
PageRight
PageUp

Open

The Open event has different arguments for different objects:

Object	See
Application	Syntax 1
Window	Syntax 2

Syntax 1

For the application object

Description

Occurs when the user starts the application.

Event ID

Event ID	Objects
None	Application

Arguments

Argument	Description
commandline	String by value Additional arguments included on the command line after the name of the executable program

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
On Windows 95, 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 INFO For an example of parsing the string in *commandline*, see CommandParm on page 452.

Examples

This code is provided in the automatically generated MDI template.

```
/* Populate SQLCA from current PB.INI settings */
SQLCA.DBMS = ProfileString("pb.ini", "database", &
   "dbms", "")
SQLCA.Database = ProfileString("pb.ini", "database",&
   "database", "")
SQLCA.Userid = ProfileString("pb.ini", "database", &
   "userid", "")
SQLCA.DBPass = ProfileString("pb.ini", "database", &
   "dbpass", "")
   SQLCA.Logid = ProfileString("pb.ini", "database", &
"logid", "")
SQLCA.Logpass = ProfileString("pb.ini", "database", &
   "LogPassWord", "")
SQLCA.Servername = ProfileString("pb.ini", &
   "database", "servername", "")
SQLCA.DBParm = ProfileString("pb.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

Description

Occurs when a window is opened by one of the Open functions. The event occurs after the window has been opened but before it is displayed.

Event ID

Event ID	Objects
pbm_open	Window

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 may 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:

```
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

Event ID	Objects
pbm_other	Windows and controls that can be placed in windows

Arguments

Argument	Description
wparam	UnsignedLong by value
lparam	Long by value

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 (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

Event ID	Objects
pbm_sbnpagedown	VScrollBar

Arguments

None

Return codes

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

0 Continue processing

Usage

When the user clicks in a vertical scrollbar, nothing happens unless you have scripts that change the scrollbar's Position property. For the scrollbar arrows, use the LineUp and LineDown events; for clicks in the scrollbar background above and below the thumb, use the PageUp and PageDown events; for dragging the thumb itself, use the Moved event.

Examples

Example 1 This code in the VScrollBar's PageDown event uses a predetermined paging value stored in the instance variable ii_pagesize to change the position of the scroll box (you would need additional code to change the view of associated controls according to the scrollbar position):

```
IF This.Position > &
This.MaxPosition - ii_pagesize THEN
   This.Position = MaxPosition
ELSE
   This.Position = This.Position + ii_pagesize
END IF
RETURN 0
```

Example 2 This example changes the position of the scroll box by a predetermined page size stored in the instance variable ii_pagesize and scrolls forward through a DataWindow control 10 rows for each page:

```
long ll_currow, ll_nextrow

This.Position = This.Position + ii_pagesize
ll_currow = dw_1.GetRow()

ll_nextrow = ll_currow + 10
dw_1.ScrollToRow(ll_nextrow)
dw_1.SetRow(ll_nextrow)
```

See also

LineDown
PageLeft
PageRight
PageUp

PageLeft

Description

Occurs when the open space to the left of the scroll box is clicked.

Event ID

Event ID	Objects
pbm_sbnpageup	HScrollBar

Arguments

None

Return codes

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

0 Continue processing

Usage

When the user clicks in a horizontal scrollbar, nothing happens unless you have scripts that change the scrollbar's Position property. For the scrollbar arrows, use the LineLeft and LineRight events; for clicks in the scrollbar 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 scrollbar (the page size is stored in the instance variable ii_pagesize):

```
IF This.Position < &
This.MinPosition + ii_pagesize THEN
   This.Position = MinPosition
ELSE
   This.Position = This.Position - ii_pagesize
END IF
```

See also

LineLeft
PageDown
PageRight
PageUp

PageRight

Description

Occurs when the open space to the right of the scroll box is clicked.

Event ID

Event ID	Objects
pbm_sbnpagedown	HScrollBar

Arguments

None

Return codes

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

0 Continue processing

Usage

When the user clicks in a horizontal scrollbar, nothing happens unless you have scripts that change the scrollbar's Position property. For the scrollbar arrows, use the LineLeft and LineRight events; for clicks in the scrollbar 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 scrollbar (the page size is stored in the instance variable ii_pagesize):

```
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

Event ID	Objects
pbm_sbnpageup	VScrollBar

Arguments

None

Return codes

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

0 Continue processing

Usage

When the user clicks in a vertical scrollbar, nothing happens unless you have scripts that change the scrollbar's Position property. For the scrollbar arrows, use the LineUp and LineDown events; for clicks in the scrollbar 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 scrollbar (the page size is stored in the instance variable ii_pagesize):

```
IF This.Position < &
This.MinPosition + ii_pagesize THEN
   This.Position = MinPosition
ELSE
   This.Position = This.Position - ii_pagesize
END IF
```

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:

```
long ll_currow, ll_prevrow

This.Position = This.Position - ii_pagesize

ll_currow = dw_1.GetRow( )
ll_prevrow = ll_currow - 10
dw_1.ScrollToRow(ll_prevrow)
dw_1.SetRow(ll_prevrow)
```

See also

LineUp
PageDown
PageLeft
PageRight

PictureSelected

Description

Occurs when the user selects a bitmap in the RichTextEdit control by double-clicking it or pressing ENTER after clicking it.

Event ID

Event ID	Objects
pbm_renpictureselected	RichTextEdit

Arguments

None

Return codes

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

0 Continue processing

Examples

When the user double-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:

```
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

Event ID	Objects
pbm_pipeend	Pipeline

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.

FOR INFO 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:

```
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

Event ID	Objects
pbm_pipemeter	Pipeline

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 INFO 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:

```
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

Event ID	Objects
pbm_pipestart	Pipeline

Arguments

None

Return codes

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

0 Continue processing

Usage

You can use the PipeStart event to check the status of pipeline processing.

The Start and Repair functions initiate pipeline processing.

FOR INFO 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

PrintEnd

Description

Occurs when the printing of a DataWindow or DataStore ends.

Event ID

Event ID	Objects
pbm_dwnprintend	DataWindow or DataStore

Arguments

Argument	Description
pagesprinted	Long by value (the total number of pages that were printed)

Return codes

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

0 Continue processing

Examples

This statement displays the number of pages that were printed after the Print function was called to print the contents of the DataWindow control:

```
st_1.Text = String(pagesprinted) &
   + " page(s) have been printed."
```

See also

PrintPage
PrintStart

PrintFooter

Description

Occurs when the footer of a page of the document in the RichTextEdit control is about to be printed.

Event ID

Event ID	Objects
pbm_renprintfooter	RichTextEdit

Arguments

Argument	Description
currentpage	Long by value (the number of the page being printed)
totalpages	Long by value (the total number of pages in the document) If the RichTextEdit control is sharing data with a DataWindow, *totalpages* is the total number of pages in a document instance
currentrow	Long by value If the RichTextEdit control is sharing data with a DataWindow, then the number of the row associated with the document instance currently being printed

Return codes

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

0 Continue processing
1 Do not print the header for the current page

Usage

In the PrintHeader and PrintFooter events, you can change the printed page number by setting the value of an input field that you've inserted for that purpose.

When an RichTextEdit control shares data with a DataWindow, there is a document instance for each row in the DataWindow. Each instance uses data in the associated row to populate its input fields.

FOR INFO For more information about sharing data between RichTextEdit and DataWindow controls, see ShareData on page 1360.

Examples

This statement in the PrintFooter event of a RichTextEdit control reports the progress of the print job in the StaticText st_1.

```
st_1.Text = "Printing footer on page " &
   + String(currentpage) + " of " &
   + String(totalpages) + " pages ...."
```

See also

PrintHeader

PrintHeader

Description

Occurs when the header of a page of the document in the RichTextEdit control is about to be printed.

Event ID

Event ID	Objects
pbm_renprintheader	RichTextEdit

Arguments

Argument	Description
currentpage	Long by value (the number of the page being printed)
totalpages	Long by value (the total number of pages in the document) If the RichTextEdit control is sharing data with a DataWindow, *totalpages* is the total number of pages in a document instance
currentrow	Long by value If the RichTextEdit control is sharing data with a DataWindow, the number of the row associated with the document instance currently being printed

Return codes

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

0 Continue processing
1 Do not print the header for the current page

Usage

In the PrintHeader and PrintFooter events, you can change the printed page number by setting the value of an input field that you've inserted for that purpose.

When an RichTextEdit control shares data with a DataWindow, there is a document instance for each row in the DataWindow. Each instance uses data in the associated row to populate its input fields.

FOR INFO For more information about sharing data between RichTextEdit and DataWindow controls, seeShareData on page 1360.

Examples

This statement in the PrintHeader event of a RichTextEdit control reports the progress of the print job in the StaticText st_1:

```
st_1.Text = "Printing header on page " &
   + String(currentpage) + " of " &
   + String(totalpages) + " pages ...."
```

See also

PrintFooter

PrintPage

Description

Occurs before each page of the DataWindow or DataStore is formatted for printing.

Event ID

Event ID	Objects
pbm_dwnprintpage	DataWindow or DataStore

Arguments

Argument	Description
pagenumber	Long by value (the number of the page about to be printed)
copy	Long by value (the number of the copy being printed)

Return codes

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

0 Do not skip the page
1 Skip the page

Examples

Example 1 After a script prints a DataWindow control, you can limit the number of pages to be printed by suppressing every page after page 50.

This statement in a CommandButton's Clicked event script prints the contents of the DataWindow control:

```
dw_1.Print()
```

This code in the PrintPage event of dw_1 cancels printing after reaching page 50:

```
IF pagenumber > 50 THEN This.PrintCancel()
```

Example 2 If you know every fifth page of the DataWindow contains the summary information you want, you can suppress the other pages with some arithmetic and a RETURN statement:

```
IF Mod(pagenumber / 5) = 0 THEN
   RETURN 0
ELSE
   RETURN 1
END IF
```

See also

PrintEnd
PrintStart

PrintStart

Description

Occurs when the printing of the DataWindow or DataStore starts.

Event ID

Event ID	Objects
pbm_dwnprintstart	DataWindow or DataStore

Arguments

Argument	Description
pagesmax	Long by value (the total number of pages that will be printed, unless pages are skipped)

Return codes

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

0 Continue processing

Usage

To skip printing some of the pages in the DataWindow or DataStore, write a script for the PrintPage event.

See also

PrintEnd
PrintPage

PropertyChanged

Description

Occurs after the OLE server changes the value of a property of the OLE object.

Event ID

Event ID	Objects
None	OLE

Arguments

Argument	Description
propertyname	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

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 send for all bindable properties, no matter how the property is being changed.

The PropertyChanged event occurs after the property's value has changed. You can obtain the new value through the automation interface. The change can no longer be canceled. If you want to cancel a change, write a script for the PropertyRequestEdit event.

See also

DataChange
PropertyRequestEdit
Rename
ViewChange

PropertyRequestEdit

Description

Occurs when the OLE server is about to change the value of a property of the object in the OLE control.

Event ID

Event ID	Objects
None	OLE

Arguments

Argument	Description
propertyname	String by value (the name of the property whose value changed) If *propertyname* is an empty string, a more general change occurred, such as changes to more than one property
cancelchange	Boolean by reference Whether the change will be canceled. Values are: ♦ FALSE — (Default) The change is allowed ♦ TRUE — The change is canceled

Return codes

None (do not use a RETURN statement)

Usage

Property change notifications are not supported by all OLE servers. The PropertyRequestEdit and PropertyChanged events only occur when the server supports these notifications.

Property notifications are not sent when the object is being created or loaded. Otherwise, notifications are send 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 via the automation interface and save it. To cancel the change, set the *cancelchange* argument to TRUE.

See also

DataChange
PropertyChanged
Rename
ViewChange

RButtonDown

The RButtonDown event has different arguments for different object:

Object	See
Controls and windows, except RichTextEdit	Syntax 1
RichTextEdit control	Syntax 2

Syntax 1

For controls and windows, except RichTextEdit

Description

For a window, occurs when the right mouse button is pressed in an unoccupied area of the window (any area with no visible, enabled object). The window event will occur if the cursor is over an invisible or disabled control.

For a control, occurs when the right mouse button is pressed on the control.

Event ID

Event ID	Objects
pbm_rbuttondown	Windows and controls that can be placed on a window, except RichTextEdit

Arguments

Argument	Description
flags	UnsignedLong by value (the modifier keys and mouse buttons that are pressed) Values are: ♦ 1 — Left mouse button ♦ 2 — Right mouse button ♦ 4 — SHIFT key ♦ 8 — CTRL key ♦ 16 — Middle mouse button In the RButtonDown event, the right mouse button is always pressed, so 2 is always summed in the value of *flags* For an explanation of *flags*, see Syntax 2 of MouseMove on page 302
xpos	Integer by value (the distance of the pointer from the left edge of the window's workspace in PowerBuilder units)

Argument	Description
ypos	Integer by value (the distance of the pointer from the top of the window's workspace in PowerBuilder units)

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 popup menu at the cursor position. Menu4 was created in the Menu painter and includes a menu called m_language. Menu4 is not the menu for the active window and therefore needs to be created. NewMenu is an instance of Menu4 (data type Menu4):

```
Menu4 NewMenu
NewMenu = CREATE Menu4
NewMenu.m_language.PopMenu(xpos, ypos)
```

In an MDI application, the arguments for PopMenu need to specify coordinates relative to the MDI frame:

```
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

Event ID	Objects
pbm_renrbuttondown	RichTextEdit

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 popup menu is displayed instead, and the RButtonDown event does not occur.

You can use the RButtonDown event to implement your own popup menu.

See also

Clicked
RButtonDown

RButtonUp

Description

Occurs when the right mouse button is released.

Event ID

Event ID	Objects
pbm_renrbuttonup	RichTextEdit

Arguments

None

Return codes

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

0 Continue processing

See also

RButtonDown

RemoteExec

Description

Occurs when a DDE client application has sent a command.

Event ID

Event ID	Objects
pbm_ddeexecute	Window

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

Event ID	Objects
pbm_ddeadvise	Window

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 will trigger the RemoteHotLinkStart event in the server application window:

```
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:

```
ib_hotlink = TRUE
```

See also

HotLinkAlarm
RemoteHotLinkStop
StartServerDDE in Chapter 9
StopServerDDE in Chapter 9
SetDataDDE in Chapter 9

RemoteHotLinkStop

Description

Occurs when a DDE client application wants to end a hot link.

Event ID

Event ID	Objects
pbm_ddeunadvise	Window

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 will trigger the RemoteHotLinkStop event in the server application window:

```
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:

```
ib_hotlink = FALSE
```

See also

HotLinkAlarm
RemoteHotLinkStart
StartServerDDE in Chapter 9
StopServerDDE in Chapter 9
SetDataDDE in Chapter 9

RemoteRequest

Description

Occurs when a DDE client application requests data.

Event ID

Event ID	Objects
pbm_dderequest	Window

Arguments

None

Return codes

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

0 Continue processing

See also

RemoteExec
RemoteSend

RemoteSend

Description

Occurs when a DDE client application has sent data.

Event ID

Event ID	Objects
pbm_ddepoke	Window

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

Event ID	Objects
pbm_omnrename	OLE

Arguments

None

Return codes

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

0 Continue processing

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

Event ID	Objects
pbm_dwnresize	DataWindow
pbm_size	Window

Arguments

Argument	Description
sizetype	UnsignedLong by value ♦ 0 — (SIZE_RESTORED) The window or DataWindow has been resized, but it was not minimized or maximized. The user may have dragged the borders or a script may have called the Resize function ♦ 1 — (SIZE_MINIMIZED) The window or DataWindow has been minimized ♦ 2 — (SIZE_MAXIMIZED) The window or DataWindow has been maximized ♦ 3 — (SIZE_MAXSHOW) This window is a popup window and some other window in the application has been restored to its former size (does not apply to DataWindow controls) ♦ 4 — (SIZE_MAXHIDE) This window is a popup window and some other window in the application has been maximized (does not apply to DataWindow controls)
newwidth	Integer by value (the width of the client area of a window or DataWindow control in PowerBuilder units)
newheight	Integer by value (the height of the client area of a window or DataWindow control in PowerBuilder units)

Return codes

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

0 Continue processing

RetrieveEnd

Description

Occurs when the retrieval for the DataWindow or DataStore is complete.

Event ID

Event ID	Objects
pbm_dwnretrieveend	DataWindow or DataStore

Arguments

Argument	Description
rowcount	Long by value (the number of rows that were retrieved)

Return codes

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

0 Continue processing

Examples

This MessageBox displayed in the RetrieveEnd event script tells the user the number of rows just retrieved:

```
MessageBox("Total rows retrieved", String(rowcount))
```

See also

RetrieveRow
RetrieveStart
SQLPreview
UpdateStart

RetrieveRow

Description

Occurs after a row has been retrieved.

Event ID

Event ID	Objects
pbm_dwnretrieverow	DataWindow or DataStore

Arguments

Argument	Description
row	Long by value (the number of the row that was just retrieved)

Return codes

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

0 Continue processing
1 Stop the retrieval

Usage

If you want to guard against potentially large queries, you can have code in the RetrieveRow event check the row argument and decide whether the user has reached a maximum limit. When *row* exceeds the limit, you can return 1 to abort the retrieval (in which case the retrieval cannot be resumed).

A script in the RetrieveRow event (even a comment) can significantly increase the time it takes to complete a query.

Obsolete functions Instead of calling SetActionCode, use the RETURN statement with a return code instead.

Examples

This code for the RetrieveRow event aborts the retrieval after 250 rows have been retrieved.

```
IF row > 250 THEN
   MessageBox("Retrieval Halted", &
   "You have retrieved 250 rows, the allowed &
   maximum.")
   RETURN 1
ELSE
   RETURN 0
END IF
```

See also

RetrieveEnd
RetrieveStart
SQLPreview
UpdateStart

RetrieveStart

Description

Occurs when the retrieval for the DataWindow or DataStore is about to begin.

Event ID

Event ID	Objects
pbm_dwnretrievestart	DataWindow or DataStore

Arguments

None

Return codes

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

0 Continue processing
1 Do not perform the retrieval
2 Do not reset the rows and buffers before retrieving data

Usage

A return code of 2 prevents previously retrieved data from being cleared, allowing the current retrieval process to append new rows to the old data.

Obsolete functions Instead of calling SetActionCode, use the RETURN statement with a return code instead.

Examples

Example 1 This statement in the RetrieveStart event prevents a reset from taking place (rows will be added to the end of the previously retrieved rows):

```
RETURN 2
```

Example 2 This statement in the RetrieveStart event aborts the retrieval:

```
RETURN 1
```

Example 3 This code allows rows to be retrieved only when a user has an ID between 101 and 200 inclusive (the ID was stored in the instance variable il_id_number when the user started the application); all other IDs cannot retrieve rows:

```
CHOOSE CASE il_id_number
    CASE IS < 100
        RETURN 1

    CASE 101 to 200
        RETURN 0

    CASE IS > 200
        RETURN 1
END CHOOSE
```

See also

RetrieveEnd
RetrieveRow
SQLPreview
UpdateStart

RightClicked

The RightClicked event has different arguments for different objects:

Object	See
ListView and Tab control	Syntax 1
TreeView control	Syntax 2

Syntax 1

For ListView and Tab controls

Description

Occurs when the user clicks the right mouse button on the ListView control or the tab portion of the Tab control.

Event ID

Event ID	Objects
pbm_lvnrclicked	ListView
pbm_tcnrclicked	Tab

Arguments

Argument	Description
index	Integer by value (the index of the item or tab the user clicked)

Return codes

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

0 Continue processing

Usage

When the user clicks in the display area of the Tab control, the tab page user object gets an RButtonDown event rather than a RightClicked event for the Tab control.

Examples

This example for the RightClicked event of a ListView control displays a popup menu when the user clicks the right mouse button:

```
// 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 pointer position
m_lv_popmenu.m_entry.PopMenu(Parent.PointerX(), &
   Parent.PointerY())
```

See also

Clicked
RightDoubleClicked

Syntax 2

For TreeView controls

Description

Occurs when the user clicks the right mouse button on the TreeView control.

Event ID

Event ID	Objects
pbm_tvnrclicked	TreeView

Arguments

Argument	Description
handle	Long by value (the handle of the item the user clicked)

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 popup menu when the user clicks the right mouse button:

```
// 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:

Object	See
ListView and Tab control	Syntax 1
TreeView control	Syntax 2

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

Event ID	Objects
pbm_lvnrdoubleclicked	ListView
pbm_tcnrdoubleclicked	Tab

Arguments

Argument	Description
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:

```
integer li_rtn

// Delete the item
li_rtn = This.DeleteItem(index)

IF li_rtn = 1 THEN
   This.Arrange( )
ELSE
   MessageBox("Error", Deletion failed!")
END IF
```

See also

DoubleClicked
RightClicked

Syntax 2 For TreeView controls

Description

Occurs when the user double-clicks the right mouse button on the TreeView control.

Event ID

Event ID	Objects
pbm_tvnrdoubleclicked	TreeView

Arguments

Argument	Description
handle	Long by value (the handle of the item the user double-clicked)

Return codes

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

0 Continue processing

Examples

This example toggles between displaying and hiding TreeView lines when the user right-double-clicks on the control:

```
IF This.HasLines = FALSE THEN
   This.HasLines = TRUE
   This.LinesAtRoot = TRUE
ELSE
   This.HasLines = FALSE
   This.LinesAtRoot = FALSE
END IF
```

See also

DoubleClicked
RightClicked

RowFocusChanged

Description

Occurs when the current row changes in the DataWindow.

Event ID

Event ID	Objects
pbm_dwnrowchange	DataWindow

Arguments

Argument	Description
currentrow	Long by value (the number of the row that has just become current)

Return codes

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

0 Continue processing

Usage

The SetRow function, as well as user actions, can trigger the RowFocusChanged and ItemFocusChanged events.

Examples

This example displays the current row number and the total number of rows in a SingleLineEdit:

```
sle_1.Text = "Row " + String(currentrow) &
    + " of " + String(This.RowCount())
```

See also

ItemFocusChanged

RowFocusChanging

Description

Occurs when the current row is about to change in the DataWindow. (The current row of the DataWindow is not necessarily the same as the current row in the database.)

The RowFocusChanging event occurs just before the RowFocusChanged event.

Event ID

Event ID	Objects
pbm_dwnrowchanging	DataWindow

Arguments

Argument	Description
currentrow	Long by value The number of the row that is current (before the row is deleted or its number changes). If the DataWindow object is empty, *currentrow* is 0 to indicate there is no current row
newrow	Long by value The number of the row that is about to become current. If the new row is going to be an inserted row, *newrow* is 0 to indicate that it does not yet exist

Return codes

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

0 Continue processing (setting the current row)
1 Prevent the current row from changing

Usage

Typically the RowFocusChanging event is coded to respond to a mouse click or keyboard action that would change the current row in the DataWindow object. The following functions can also trigger the RowFocusChanging event, as well as the RowFocusChanged and ItemFocusChanged events, when the action changes the current row:

SetRow
Retrieve
RowsCopy
RowsMove
DeleteRow
DiscardRows

In these cases, the RowFocusChanging event script can prevent the changing of the DataWindow object current row only. The script cannot prevent the data from being changed (for example, the rows still get moved).

Examples

This example displays a message alerting you that changes have been made in the window dw_detail which will be lost if the row focus is changed to the window dw_master.

```
IF dw_detail.DeletedCount() > 0 OR &
   dw_detail.ModifiedCount() > 0 THEN
   MessageBox("dw_master", "Changes will be lost &
      in Detail")
ELSE Return 0
END IF
```

See also

ItemFocusChanged
RowFocusChanged

Save

Description

Occurs when the server application notifies the control that the data has been saved.

Event ID

Event ID	Objects
pbm_omnsave	OLE

Arguments

None

Return codes

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

0 Continue processing

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:

```
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

ScrollHorizontal

Description

Occurs when user scrolls left or right in the DataWindow with the TAB or arrow keys or the scrollbar.

Event ID

Event ID	Objects
pbm_dwnhscroll	DataWindow

Arguments

Argument	Description
scrollpos	Long by value The distance in the current units for the DataWindow object of the scroll box from the left end of the scrollbar in the DataWindow (if the DataWindow is split, in the pane being scrolled)
pane	Integer by value The number of the pane being scrolled (when the DataWindow is split with two scrollbars, there are two panes). Values are: ♦ 1 — The left pane (if the scrollbar is not split, the only pane) ♦ 2 — The right pane

Return codes

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

0 Continue processing

Examples

This example displays the customer ID of the current row (the cust_id column) in a SingleLineEdit control when the pane being scrolled is pane 1 and the position is greater than 100:

```
string ls_id
ls_id = ""
IF pane = 1 THEN
   IF scrollpos > 100 THEN
      ls_id = &
         String(dw_1.Object.Id[dw_1.GetRow( )])
   END IF
END IF
```

```
sle_message.Text = ls_id

RETURN 0
```

See also

ScrollVertical

ScrollVertical

Description

Occurs when user scrolls up or down in the DataWindow with the TAB or arrow keys or the scrollbar.

Event ID

Event ID	Objects
pbm_dwnvscroll	DataWindow

Arguments

Argument	Description
scrollpos	Long by value (the distance in the current units for the DataWindow object of the scroll box from the top of the scrollbar)

Return codes

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

0 Continue processing

Examples

As the user scrolls vertically, this script displays the range of rows currently being displayed in the DataWindow:

```
long ll_numrows
string ls_firstrow, ls_lastrow

ll_numrows = dw_1.RowCount()
ls_firstrow = dw_1.Object.Datawindow.FirstRowOnPage
ls_lastrow = dw_1.Object.Datawindow.LastRowOnPage

sle_message.Text = "Rows " + ls_firstrow &
    + " through " + ls_lastrow + " of " &
    + String(ll_numrows)

RETURN 0
```

See also

ScrollHorizontal

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

Event ID	Objects
None	Menu

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:

```
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:

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

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

```
w_test.wf_SetMenuHelp("")
Timer(0)
```

See also

Clicked

SelectionChanged

The SelectionChanged event has different arguments for different objects:

Object	See
DropDownListBox, DropDownPictureListBox, ListBox, PictureListBox controls	Syntax 1
Tab control	Syntax 2
TreeView control	Syntax 3

Syntax 1

For Listboxes

Description

Occurs when an item is selected in the control.

Event ID

Event ID	Objects
pbm_cbnselchange	DropDownListBox, DropDownPictureListBox
pbm_lbnselchange	ListBox, PictureListBox

Arguments

Argument	Description
index	Integer by value (the index of the item that has become selected)

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 dropdown 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 (it is a MultiSelect ListBox so *index* doesn't matter). The script checks all the items to see if they are selected:

```
integer itemcount,i,r
string ls_colname

gr_1.SetRedraw(FALSE)

// Clear out categories, series and data from graph
gr_1.Reset(All!)

// Loop through all selected values and
// create as many series as the user specified
FOR i = 1 to lb_value.TotalItems()
   IF lb_value.State(i) = 1 THEN
      ls_colname = lb_value.Text(i)

      // Call window function to set up the graph
      wf_set_a_series(ls_colname, ls_colname, &
         lb_category.text(1))
   END IF

NEXT

gr_1.SetRedraw(TRUE)
```

See also

Clicked

Syntax 2 For Tab controls

Description

Occurs when a tab is selected.

Event ID

Event ID	Objects
pbm_tcnselchanged	Tab

Arguments

Argument	Description
oldindex	Integer by value (the index of the tab that was previously selected)
newindex	Integer by value (the index of the tab that has become selected)

Return codes

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

0 Continue processing

Usage

The SelectionChanged event occurs when the Tab control is created and the initial selection is set.

See also

Clicked
SelectionChanging

Syntax 3 For TreeView controls

Description

Occurs when the item is selected in a TreeView control.

Event ID

Event ID	Objects
pbm_tvnselchanged	TreeView

Arguments

Argument	Description
oldhandle	Long by value (the handle of the previously selected item)
newhandle	Long by value (the handle of the currently selected item)

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:

```
TreeViewIteml_tvinew, l_tviold

//get the treeview item that was the old selection
This.GetItem(oldhandle, l_tviold)

//get the treeview item that is currently selected
This.GetItem(newhandle, l_tvinew)

//Display the labels for the two items in sle_get
sle_get.Text = "Selection changed from " &
    + String(l_tviold.Label) + " to " &
    + String(l_tvinew.Label)
```

See also

Clicked
SelectionChanging

SelectionChanging

The SelectionChanging event has different arguments for different objects:

Object	See
Tab control	Syntax 1
TreeView control	Syntax 2

Syntax 1

For Tab controls

Description

Occurs when another tab is about to be selected.

Event ID

Event ID	Objects
pbm_tcnselchanging	Tab

Arguments

Argument	Description
oldindex	Integer by value (the index of the currently selected tab)
newindex	Integer by value (the index of the tab that is about to be selected)

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, then the controls on the page do not exist yet and you can't 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:

```
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

Event ID	Objects
pbm_tvnselchanging	TreeView

Arguments

Argument	Description
oldhandle	Long by value (the handle of the currently selected item)
newhandle	Long by value (the handle of the item that is about to be selected)

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:

```
TreeViewItem l_tvinew, l_tviold

// Get TreeViewItem that was the old selection
This.GetItem(oldhandle, l_tviold)

// Get TreeViewItem that is currently selected
This.GetItem(newhandle, l_tvinew)

//Display the labels for the two items in display
sle_status.Text = "Selection changed from " &
   + String(l_tviold.Label) + " to " &
   + String(l_tvinew.Label)
```

See also

Clicked
SelectionChanged

Show

Description

Occurs just before the window is displayed.

Event ID

Event ID	Objects
pbm_showwindow	Window

Arguments

Argument	Description
show	Boolean by value (whether the window is being shown) The value is always TRUE.
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 popup window is being hidden ♦ 3 — The window's parent is also being displayed or maximized or a popup 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

Sort

The Sort event has different arguments for different objects:

Object	See
ListView control	Syntax 1
TreeView control	Syntax 2

Syntax 1

For ListView controls

Description

Occurs for each comparison when the TreeView is being sorted.

Event ID

Event ID	Objects
pbm_lvnsort	ListView

Arguments

Argument	Description
index1	Integer by value (the index of one item being compared during a sorting operation)
index2	Integer by value (the index of the second item being compared)
column	Integer by value (the number of the column containing the items being sorted)

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 be selecting one of the return codes.

You typically use the Sort event when you want to sort ListView items based on multiple criteria such as a PictureIndex and Label.

The Sort event occurs if you call the Sort event, or when you call the Sort function using the UserDefinedSort! argument.

Examples

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

```
ListViewItem lvi, lvi2

This.GetItem(index1, lvi)
This.GetItem(index2, lvi2)

IF lvi.PictureIndex > lvi2.PictureIndex THEN
   RETURN 1
ELSEIF lvi.PictureIndex < lvi2.PictureIndex THEN
   RETURN -1
ELSEIF lvi.label > lvi2.label THEN
   RETURN 1
ELSEIF lvi.label < lvi2.label THEN
   RETURN -1
ELSE
   RETURN 0
END IF
```

Syntax 2 — For TreeView controls

Description

Occurs for each comparison when the TreeView is being sorted.

Event ID

Event ID	Objects
pbm_tvnsort	TreeView

Arguments

Argument	Description
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 be selecting one of the return codes.

You typically use the Sort event when you want to sort TreeView items based on multiple criteria such as a PictureIndex and Label.

The Sort event occurs if you call the Sort event, or when you call the Sort function using the UserDefinedSort! argument.

Examples

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

```
TreeViewItem tvi, tvi2

This.GetItem(handle1, tvi)
This.GetItem(handle2, tvi2)

IF tvi.PictureIndex > tvi2.PictureIndex THEN
   RETURN 1
ELSEIF tvi.PictureIndex < tvi2.PictureIndex THEN
   RETURN -1
ELSEIF tvi.Label > tvi2.Label THEN
   RETURN 1
ELSEIF tvi.Label < tvi2.Label THEN
   RETURN -1
ELSE
   RETURN 0
END IF
```

SQLPreview

Description

Occurs immediately before a SQL statement is submitted to the DBMS. Functions that trigger DBMS activity are Retrieve, Update, and ReselectRow.

Event ID

Event ID	Objects
pbm_dwnsql	DataWindow

Arguments

Argument	Description
request	SQLPreviewFunction by value (the function that initiated the database activity) Values are: ♦ PreviewFunctionReselectRow! — ReselectRow function ♦ PreviewFunctionRetrieve! — Retrieve function ♦ PreviewFunctionUpdate! — Update function
sqltype	SQLPreviewType by value (the type of SQL statement being sent to the DBMS) Values are: ♦ PreviewDelete! — A DELETE statement ♦ PreviewInsert! — An INSERT statement ♦ PreviewSelect! — A SELECT statement ♦ PreviewUpdate! — An UPDATE statement
sqlsyntax	String by value (the full text of the SQL statement)
buffer	DWBuffer by value (the buffer containing the row involved in the database activity) Values are: ♦ Delete! — The delete buffer (data that has been deleted from the DataWindow) ♦ Filter! — The filter buffer (data that has been filtered out) ♦ Primary! — The primary buffer (data that has not been deleted or filtered out)
row	Long by value (the number of the row involved in the database activity, that is, the row being updated, selected, inserted, or deleted)

Return codes

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

0 Continue processing
1 Stop processing
2 Skip this request and execute the next request

Usage

Obsolete functions Information formerly provided by GetSQLPreview and GetUpdateStatus is available in the arguments *sqlsyntax* , *row*, and *buffer*.

Some uses for the *sqlsyntax* argument are:

- Changing the SQL to be executed (you can get the value of *sqlsyntax*, modify it, and call SetSQLPreview)
- Keeping a record (you can write the SQL statement to a log file)

GetSQLPreview and binding
When binding is enabled for your database, the SQL returned in the GetSQLPreview event may not be complete—the input arguments are not replaced with the actual values. For example, when binding is enabled, GetSQLPreview might return the following SQL statement:

```
INSERT INTO "cust_order" ( "ordnum", "custnum",
"duedate", "balance" ) VALUES ( ?, ?, ?, ? )
```

When binding is disabled, it returns:

```
INSERT INTO "cust_order" ( "ordnum", "balance",
"duedate", "custnum" ) VALUES ( '12345', 900,
'3/1/94', '111' )
```

If you require the complete SQL statement for logging purposes, you should disable binding in your DBMS.

FOR INFO For more information about binding, see *Connecting to Your Database.*

If the row that caused the error is in the Filter buffer, you must unfilter it if you want the user to correct the problem.

Reported row number
The row number stored in *row* is the number of the row in the buffer, not the number the row had when it was retrieved into the DataWindow object.

Examples

This statement in the SQLPreview event sets the current SQL string for the DataWindow dw_1:

```
dw_1.SetSQLPreview( &
```

```
"INSERT INTO billings VALUES(100, " + &
String(Current_balance) + ")")
```

See also

RetrieveStart
UpdateEnd
UpdateStart

SystemError

Description

Occurs when a serious execution time error occurs (such as trying to open a nonexistent window).

Event ID

Event ID	Objects
None	Application

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 INFO 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.

After a SystemError event occurs, it is not a good idea to continue running the application. Use the event script to clean up and disconnect from the DBMS and then exit the application.

Examples

This statement in the SystemError event halts the application immediately:

```
HALT CLOSE
```

See also

Error

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

Event ID	Objects
pbm_syskeydown	Window

Arguments

Argument	Description
key	KeyCode by value A value of the KeyCode enumerated data type indicating the key that was pressed, for example, KeyA! or KeyF1!
keyflags	UnsignedLong by value (the modifier keys that were pressed with the key) The only modifier key is: 1 SHIFT key

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.

On Macintosh
The SystemKey event is not available on the Macintosh.

Examples

This example displays the name of the key that was pressed with the ALT key:

```
string ls_key

CHOOSE CASE key

CASE KeyF1!
   ls_key = "F1"
CASE KeyA!
   ls_key = "A"
CASE KeyF2!
   ls_key = "F2"
END CHOOSE
```

This example causes a beep if the user presses ALT+SHIFT+F1.

```
IF keyflags = 1 THEN
   IF key = KeyF1 THEN
      Beep(1)
   END IF
END IF
```

See also

Key

Timer

Description

Occurs when a specified number of seconds elapses after the Start or Timer function has been called.

Event ID

Event ID	Objects
pbm_timer	Timing or Window

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 even connects to the database:

```
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:

```
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:

```
ds_datastore = CREATE datastore
```

The timing object's Timer event calls an object-level function that refreshes the datastore:

```
refresh_custlist()
```

Here's refresh_custlist():

```
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:

```
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:

```
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")
```

ToolbarMoved

Description

Occurs in an MDI frame window when the user moves any FrameBar or SheetBar.

Event ID

Event ID	Objects
pbm_tbnmoved	Window

Arguments

None

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.

UpdateEnd

Description

Occurs when all the updates to the database from the DataWindow (or DataStore) are complete.

Event ID

Event ID	**Objects**
pbm_dwnupdateend	DataWindow or DataStore

Arguments

Argument	Description
rowsinserted	Long by value (the number of rows inserted)
rowsupdated	Long by value (the number of rows updated)
rowsdeleted	Long by value (he number of rows deleted)

Return codes

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

0 Continue processing

See also

RetrieveStart
SQLPreview
UpdateStart

UpdateStart

Description

Occurs after a script calls the Update function and just before changes in the DataWindow or DataStore are sent to the database.

Event ID

Event ID	Objects
pbm_dwnupdatestart	DataWindow or DataStore

Arguments

None

Return codes

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

0 Continue processing
1 Do not perform the update

See also

RetrieveStart
SQLPreview
UpdateEnd

ViewChange

Description

Occurs when the server application notifies the control that the view shown to the user has changed.

Event ID

Event ID	Objects
pbm_omnviewchange	OLE

Arguments

None

Return codes

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

0 Continue processing

See also

DataChange
PropertyRequestEdit
PropertyChanged
Rename

Index

-- (assignment shortcut) 129
- *see* dashes

Symbols

! (enumerated value) 34
& *see* ampersand
* (multiplication) 74
+ (addition) 74
++, += (assignment shortcuts) 129
/ (division) 74
// (comments) 4
/= (assignment shortcut) 129
; (SQL) 19
< (less than) 76
<= (less than or equal) 76
<> (not equal) 76
= (assignment) 43
= (relational) 76
> (greater than) 76
>= (greater than or equal) 76
? (dynamic SQL) 184, 186, 189
^ (exponentiation) 74
{} 3
~ *see* tilde
' *see* quotes

A

Abs function 382
absolute value 382
AcceptText function
  about 383
  calling from Update 1456
access levels
  functions 63
  group label 48
  variables 45
action code 1234
Activate event 200
Activate function 385
active sheet 990
active window 1035
AddCategory function 387
AddColumn function 389
AddData function 391
AddItem function 394
addition 74
AddLargePicture function 400
AddPicture function 401
address keyword 1444
address, mail 896, 909, 911
AddSeries function 403
AddSmallPicture function 405
AddStatePicture function 406
ALLBASE 950
AllowEdit property 1215
ampersand (&) 19
ancestor
  functions 121
  hierarchy 429
  objects 91
  return values from events 121, 197
  script, calling 131
AND operator 76
angle
  calculating cosine 469
  calculating sine 1376
  calculating tangent 1416
  converting to/from radians 1031
ANSI 1424, 1430
Any data type 29
API and database handles 501
appending 1126
application
  closing DDE channel 442
  connecting to 454, 456, 458, 461
  elapsed time 470

- exporting object as syntax 879
- handle 621, 775
- listing objects 877
- posting messages 1047
- recreating objects from syntax 881
- remote 1348
- restarting 1145
- retrieving arguments 452
- running 1164
- server 1395, 1403
- terminating 147
- yielding to 1474

application name 1393, 1395, 1403
Application objects, SetTransPool function 1354
Arabic functions
- IsAllArabic 847
- IsAnyArabic 849
- IsArabic 851
- IsArabicAndNumbers 852

arguments
- command line 452
- for events 196
- functions and events 112
- hot link 1393, 1400
- in SetSQLSelect function 1330
- objects 112
- retrieval 1146
- server application 1395, 1403

arithmetic operators 74
Arrange function 407
ArrangeOpen enumerated data type 990
ArrangeSheets function 408
ArrangeTypes enumerated data type 408
array functions
- LowerBound 895
- UpperBound 1463

arraylists 58
arrays
- about 51
- assigning values 56, 58, 128
- chars and strings 83
- copying 128
- default values 54
- errors 59
- example 410
- initializing 58
- input parameter for dynamic SQL 1270
- mailRecipient 896
- message ID 898
- passing as arguments 113
- stream 1104, 1471
- variable-size 55

arrow pointer 1301
Asc function 410
ASCII values
- converting characters to 410
- of nonprinting characters 1087

assignment
- arrays 54, 56, 58
- overflow 81
- shortcut operators 129
- statements 128

asterisk in text patterns 920
AttachmentFile property 907
audio (beep) 412
AutoCommit 1411
Autoinstantiate setting 92
automation 1239, 1241, 1243
Autosize Height property 1267
axis, graphs
- categories 387, 422, 504, 808
- inserting data 812

B

back quote 131
background color, graphs
- data points 660, 1261
- series 741, 1317

background layer of DataWindow 1305
backslash in text patterns 919
backspace, specifying 9
bands, DataWindow
- locating 625
- moving objects to 1305
- reporting on 524
- setting row height 1267

BAT file 1164
batch applications 1049
beam pointer 1301
Beep function 412

BeginDrag event 201
BeginLabelEdit event 205
BeginRightDrag event 207
binding 751
birth dates 1473
bitmaps
- assigning to picture control 1300
- deleting and adding 941
- in rich text 835
- printing 1066
- retrieving from clipboard 435
- under pointer 725

blob data type 24
Blob function 413
blob functions
- Blob 413
- BlobEdit 414
- BlobMid 415
- Len 869

BlobEdit function 414
BlobMid function 415
blobs
- assigning to picture control 1300
- converting to string 413, 1404
- declaring 40
- extracting values from 483, 488, 503, 840, 893, 1106, 1419
- inserting data into 414
- reading streams into 1104
- selecting from database 174
- setting up columns 968
- updating 176
- writing to stream 1471

boolean data type 24
border
- determining distance from 1033, 1034
- determining style 627
- printing 1079, 1082, 1084
- setting style, for columns 1245

bottom layer of DataWindow 1305
bound 895, 1463
Box border style 627
brackets in text patterns 920
breaks 774
buffer, DataWindow
- copying rows 1160
- editing items 756
- moving rows 1162
- of updated row 766
- retrieving data 694, 699, 702, 707, 709
- returning modified rows 721
- setting values of rows and columns 1280
- sharing data 1360, 1363

BuildModel function 417
ButtonClicked event 210
ButtonClicking event 211

C

C functions
- decoding returned values 843, 844
- passing values to 892

CALL statement
- about 131
- not using 196

Cancel button 930
Cancel function 420
cancellation
- allowing 1474
- of edits 1454
- of pipeline object 420
- of printing 1068
- of row retrieval 494

CanUndo function 421
capitalization
- in category names 387, 808
- in series names 403
- lowercase 894
- uppercase 1462

caret in text patterns 919
carriage return
- in INI files 1099
- specifying 9

cascaded windows, arranging sheets 408
cascading opened windows 990
case sensitivity, comparisons 76
categories, graphs
- adding data values to series 387, 392
- adding to a series 387
- clicked 965
- counting 422

- deleting 504, 1133
- identifying 422, 423
- importing data 785, 790, 795
- InsertCategory function 387
- inserting 808
- new 387

CategoryCount function 422
CategoryName function 423
Ceiling function 424
century 1473
ChangeMenu function 425
channel, DDE 442, 987
char data type
- about 24
- array 83
- converting to string 83

Char function 426
character array 1471
characters
- array 1104
- changing capitalization 894, 1462
- converting to ASCII values 410
- extracting 426, 933
- mask 1293
- matching 919
- returning rightmost 1153
- selected 1204, 1209
- selecting 1220

Check function 427
CheckBox edit style 771
Checked property 1452
child windows
- obtaining parent 1023
- opening 972, 1014
- retrieving data for 631

CHOOSE CASE statement 132
class
- defined 88
- of object 429
- OLE 810

class hierarchy 32
class user objects 89
ClassDefinition objects, FindMatchingFunction 602
ClassList function 428
ClassName function 429
Clear function 432
clearing text 432
ClearValues function 434
Clicked event 212, 636, 637, 859, 966
clipboard
- contents as replacement text 1128
- copying 465
- cutting 478
- importing data from 784
- pasting and linking 1027
- pasting from 1025
- retrieving and replacing contents 435
- saving DataWindow to 1168

Clipboard function 435
CLOSE Cursor statement 158
Close event 220, 438, 1145
Close function 438
CLOSE Procedure statement 159
CloseChannel function 442
CloseQuery event 222, 438
CloseTab function 444
CloseUserObject function 446
CloseWithReturn function 448
closing
- DDE channel 442
- print job 1071
- windows 438

code
- generating DataWindow 1410
- object 879
- reusing 1050

code table 434, 771, 1358
cold link 555, 730, 988, 1310
CollapseItem function 451
colors
- and edit masks 1293
- changing DataWindow object 941, 944
- data point 660, 1137, 1260
- red, green, and blue components of 1151
- series 740, 1317
- supported 676
- table of standard colors 1151

column headings
- when importing data from files 789
- when inserting a string 794

ColumnClick event 224
columns

clicked 636
computed 1329
current 638, 641, 1249
deleting values 434
determining border style 627
determining insertion point position 1041
format of 685, 1277
in list 811
initializing 836
modification status of 705, 1286
pasting text into 1026
properties of 524, 527
reading from database 1130
replacing text 1333
retrieving dates from 694, 697
retrieving from buffer 707, 709
retrieving numbers from 699, 702
setting border style 1245
setting tab order 1332
setting to read-only 1332
sharing data 1360
under pointer 725
updating 1456
validation rule of 756, 770, 1356
values of 771, 1280
COM file 1164
comma 1325
command button, activating OLE object 968
command line, retrieving arguments 452
CommandParm function 452
commands
getting from DDE client 642
receiving form DDE application 1143
comments
in library 873
using 4
COMMIT statement 160
comparing
numbers 923, 936
strings 76
comparison 839
composite reports
no filtering 578
no sorting 1378
computer
beeping 412
reporting CPU time 470
concatenation, strings 78
condensed mode 1087
configuration settings
reading 1097, 1099
saving 1306
CONNECT statement 161
Connection objects
ConnectToServer function 464
CreateInstance function 474
DisconnectServer function 537
GetServerInfo function 747
RemoteStopConnection function 1121
RemoteStopListening function 1123
ConnectionBegin event 226
ConnectionEnd event 228
ConnectionInfo objects
GetServerInfo function 747
connections
specifying settings 1348
to OLE object 454, 456
ConnectToNewObject function 454
ConnectToNewRemoteObject function 456
ConnectToServer function 464
constants
about 36
assigning values 43
declaring 36, 50
where to declare 36
Constructor event 229
ContextInformation objects
GetCompanyName function 645
GetFixesVersion function 682
GetHostObject function 688
GetMajorVersion function 714
GetMinorVersion function 717
GetName function 719
GetShortName function 749
GetVersionName function 773
ContextKeyword objects, GetContextKeywords function 646
context-sensitive Help 1371
continuation character 19
CONTINUE statement 134
continuous line style
setting for data points 1263

setting for series 1319
Control array 1005, 1007
control structures
CHOOSE CASE 132
DO...LOOP 140
FOR...NEXT 144
IF...THEN 148
controls
determining type 1450
dragging 543
focus of 684, 1276
hiding 778, 959
moving 959
obtaining handle 775
redrawing 1308
referencing 449
resizing 1142
yielding 1474
coordinates
ListView items 726
of print cursor 1095, 1096
of print objects 1066, 1075, 1079, 1082, 1084
Copy function 465
copying
importing from clipboard 784
range of rows 1159
to clipboard 465
CopyRTF function 467
Cos function 469
cosine 469
count
of data points in a series 480
of rows marked for deletion 508
CPU
getting information about 676
time 470
Cpu function 470
Create function 471
CREATE statement 135, 1035
CreateInstance function 474
CreatePage function 476
creating DataWindow objects 941
criteria
input 1356
sort 1325, 1377
cross mouse pointer 1301
CrosstabDialog function 477
crosstabs
and ShareData function 1362
creating from source code 1410
defining 477
obtaining message text 716
current
column 1249
row 734, 1218, 1313
row and scrolling 1186, 1189, 1191, 1194
row before inserting 836
sheet 990
cursor
and current row 1313
custom 1301
displaying popup menus 1035
hand pointer 1315
print 1061
cursors, database
closing 158
declaring 157, 162
opening 171
custom class user objects 92
Cut function 478
cutting, to clipboard 478

D

dash line style
about 1263, 1320
setting for series 1320
dashes, prohibiting in variable names 6
DashesInIdentifiers option 6
data
adding to a graph series 391, 393
clearing 1131
converting to type long 892
correcting pipeline 1124
finding in DataWindow 582
from OLE server 653
getting DDE 655
importing 784
inserting into a blob 414
obtaining from control 650
receiving from DDE application 1143

retrieving for child window or report 631
retrieving from buffers 694, 697, 699, 702, 707, 709
sending to DDE client 1256
sharing 481, 1360, 1363
to OLE server 1254
transferring 1385
validating 1356
writing to file 575
writing to stream 1471
data expressions, Any data type 31
Data Pipeline painter 420, 1386
data points
adding to a scatter graph 393
clicked 965
deleting 504
inserting 812
reporting appearance of 660
reporting explosion percent 658
resetting colors 1137
setting style 1260
value of 650, 667
data source 941, 952
data type checking and conversion functions
Asc 410
Char 426
Date 483
DateTime 487
Dec 503
Double 540
Integer 840
IsDate 853
IsNull 856
IsNumber 857
IsTime 861
Long 892
Real 1106
String 1404
Time 1419
data types
about 24
assignment 81
blob 413
date 486
determining 429
effect of operators 81
enumerated 34
external functions 66
literals 24, 25, 26, 28, 81
mismatch when pasting 1026
numeric 80
of columns 524, 528
promotion 80
promotion for function arguments 110
real 1106
setting to NULL 1296
standard 24
string 1404
system object 32
time 1419
unknown 29
windows 970
database stored procedures 153
databases
canceling changes 172
canceling row retrieval 494
commiting changes 160
communicating with 1350
connecting to 161
cursor, opening 171
deleting rows 165, 166
disconnecting from 167
fetching rows 169
handle 501
inserting rows 170
modified rows 939
on restart 1145
preventing deletion on update 1161
reading 1130
repairing 1124
reporting errors 499
retrieving data 694, 697, 699, 702, 707, 709
retrieving rows 1146
returning error codes 497
rows 508
selecting rows 173
specifying name 1348
SQL statement 751, 752, 1328, 1329
transactions 1354
transferring data between 1385
updating 175, 766, 1456
updating cursored row 178

DataChange event 231
DataModified item status
  about 721
  setting 1140, 1287
DataSource function 481
DataStore functions
  GenerateHTMLForm 617
  GetChanges 628
  GetFullState 686
  GetRowFromRowId 735
  GetRowIdFromRow 737
  GetStateStatus 754
  SaveAsAscii 1178
  SetChanges 1246
  SetFullState 1278
  SetItem 1280
  SetItemStatus 1286
  ShareData 1360
  Sort 1377
DataWindow control
  AcceptText function 383
  data and property expressions and Any 31
  for pipline errors 1386
  rows available for display 1157
DataWindow functions
  CanUndo 421
  CategoryCount 422
  CategoryName 423
  Clear 432
  ClearValues 434
  Clipboard 435
  Copy 465
  Create 471
  CrosstabDialog 477
  Cut 478
  DataCount 480
  DBCancel 494
  DBErrorCode 497
  DBErrorMessage 499
  DeletedCount 508
  DeleteRow 518
  Describe 524
  Filter 578
  FilteredCount 580
  Find 582
  FindCategory 588
  FindGroupChange 593
  FindNext 605
  FindRequired 606
  FindSeries 610
  GenerateHTMLForm 617
  GetBandAtPointer 625
  GetBorderStyle 627
  GetChanges 628
  GetChild 631
  GetClickedColumn 636
  GetClickedRow 637
  GetColumn 638
  GetColumnName 641
  GetData 650
  GetDataPieExplode 658
  GetDataStyle 660
  GetFormat 685
  GetFullState 686
  GetItemDate 694
  GetItemDateTime 697
  GetItemDecimal 699
  GetItemNumber 702
  GetItemStatus 705
  GetItemString 707
  GetItemTime 709
  GetMessageText 716
  GetNextModified 721
  GetObjectAtPointer 725
  GetRow 734
  GetRowFromRowId 735
  GetRowIdFromRow 737
  GetSelectedRow 739
  GetSeriesStyle 740
  GetSQLPreview 751
  GetSQLSelect 752
  GetStateStatus 754
  GetText 756
  GetTrans 764
  GetUpdateStatus 766
  GetValidate 770
  GetValue 771
  GroupCalc 774
  ImportClipboard 784
  ImportFile 788
  ImportString 793
  InsertRow 836

IsSelected 859
LineCount 883
ModifiedCount 939
Modify 941
ObjectAtPointer 965
OLEActivate 968
Paste 1025
PasteRTF 1029
Position 1041
Print 1057
PrintCancel 1068
ReplaceText 1128
ReselectRow 1130
Reset 1131
ResetDataColors 1137
ResetTransObject 1139
ResetUpdate 1140
Retrieve 1146
RowCount 1157
RowsCopy 1159
RowsDiscard 1161
RowsMove 1162
SaveAs 1168
SaveAsAscii 1178
Scroll 1182
ScrollNextPage 1183, 1186
ScrollPriorPage 1189
ScrollPriorRow 1191
ScrollToRow 1194
SelectedLength 1204
SelectedLine 1206
SelectedStart 1209
SelectedText 1211
SelectRow 1218
SelectText 1220
SeriesCount 1231
SeriesName 1232
SetActionCode 1234
SetBorderStyle 1245
SetChanges 1246
SetColumn 1249
SetDataPieExplode 1258
SetDataStyle 1260
SetFilter 1272
SetFormat 1277
SetFullState 1278
SetItem 1280
SetItemStatus 1286
SetPosition 1305
SetRow 1313
SetRowFocusIndicator 1315
SetSeriesStyle 1317
SetSort 1325
SetSQLPreview 1328
SetSQLSelect 1329
SetTabOrder 1332
SetText 1333
SetTrans 1348
SetTransObject 1350
SetValidate 1356
SetValue 1358
ShareData 1360
ShareDataOff 1363
Sort 1377
TextLine 1418
Undo 1454
Update 1456

DataWindow object
changing text 947
creating 471
creating from SELECT statement 1410
creating from source code 1410
deleting from libraries 875
exporting as syntax 879
listing 877
properties of 524
recreating from syntax 881

date data type 24

Date function 483

date, day, and time functions
Day 489
DayName 490
DayNumber 491
DaysAfter 492
Hour 780
Minute 937
Month 958
Now 964
RelativeDate 1117
RelativeTime 1118
Second 1197
SecondsAfter 1198

Today 1425
Year 1473
dates
checking string 853
converting to 484
DateTime data type 483, 487
day of week 490, 491
determining interval 492
getting dynamic 669, 671
in blobs 483
obtaining current 1425
obtaining day of month 489
retrieving from buffer 694, 697
DateTime data type
about 25
retrieving from buffers 697
DateTime function 487
Day function 489
DayName function 490
DayNumber function 491
DaysAfter function 492
dBase file
importing data from 788, 793
saving to 1168
DBCancel function 494
DBError event 232, 497, 499, 751, 766
DBErrorCode function 497
DBErrorMessage function 499
DBHandle function 501
DBMS
setting connection parameters 1349, 1350
timestamp support 1130
DDE channel
closing 442
requesting data 731
DDE client functions
CloseChannel 442
ExecRemote 555
GetDataDDE 655
GetDataDDEOrigin 656
GetRemote 730
OpenChannel 987
RespondRemote 1143
SetRemote 1310
StartHotLink 1393
StopHotLink 1400
DDE server functions
GetCommandDDE 642
GetCommandDDEOrigin 644
GetDataDDE 655
GetDataDDEOrigin 656
RespondRemote 1143
SetDataDDE 1256
StartServerDDE 1395
StopServerDDE 1403
DDL, executing through dynamic SQL 183, 184
Deactivate event 234
DebugBreak function 502
debugging, debug mode 944
Dec function 503
decimal data type
about 25
converting to 503
declaring 40
retrieving from buffers 699
declarations
about 36
access levels 45
arrays 51
constants 50
expressions as initial values 44
syntax 40
variables 36
where to declare 36
DECLARE Cursor statement 162
DECLARE Procedure statement 163
default values 836
definition
changing DataWindow object 941
font, for printing 1073
delete buffer
discarding rows from 1161
emptying 1140
retrieving data 694, 697, 699, 702, 707, 709
returning modified rows 721
sharing data 1360, 1363
DELETE statement 165
DELETE Where Current of Cursor statement 166
DeleteAllItems event 235
DeleteCategory function 504
DeleteColumn function 505
DeleteColumns function 506

DeleteData function 507
DeletedCount function 508
DeleteItem event 236
DeleteItem function 510
DeleteLargePicture function 514
DeleteLargePictures function 515
DeletePicture function 516
DeletePictures function 517
DeleteRow function 518
DeleteSeries function 519
DeleteSmallPicture function 520
DeleteSmallPictures function 521
DeleteStatePicture function 522
DeleteStatePictures function 523
descendant
  determining class of 429
  opening user object 997, 998, 1006, 1008
  opening window 974
  return values from events 121, 197
Describe function
  about 524
  property values 526
DESTROY statement
  about 91, 139
  ending a mail session 903
destroying DataWindow objects 941
DestroyModel function 530
Destructor event 238, 444, 446
detail bands
  locating 625
  moving objects to 1305
  setting row height 1267
diagonal fill pattern 1265, 1321
dialog
  defining crosstabs 477
  Insert Object 834
  Open File 677
  PasteSpecial 1030
  Save File 679
diamond fill pattern 1265, 1322
DIF file 1168
dimension 895
dimension of array 1463
directory, of library 877
DirList function 531
DirSelect function 533
Disable function 535
DISCONNECT statement 167, 1348
DisconnectObject function 536
DisconnectServer function 537
display format
  applying to string 1404
  of columns 685, 1277
distributed applications
  ConnectToServer function 464
  DisconnectServer function 537
  GetChanges function 628
  GetFullState function 686
  GetServerInfo function 747
  GetStateStatus function 754
  Listen function 889
  RemoteStopConnection function 1121
  RemoteStopListening function 1123
  SetChanges function 1246
  SetFullState function 1278
  SharedObjectDirectory function 1364
  SharedObjectGet function 1365
  SharedObjectRegister function 1367, 1368
  StopListening function 1402
distributed applicatons
  RemoteStopConnection function 1121
division 74, 75, 938
DLLs for external functions 63
DO...LOOP statement 140
document windows 990
dollar sign in text patterns 919
DoScript function 538
dot notation
  about 38
  instance variables 38
  structures 86
dotted line style
  setting for data points 1263
  setting for series 1320
  setting row focus indicator 1315
double colon 131
double data type 25
Double function 540
DoubleClicked event 239, 636, 637
DoubleParm property 994, 1001, 1003, 1010, 1012
DoVerb function 541
Drag function 543

DragDrop event 244
DragEnter event 250
DraggedObject function 545
dragging, TreeView items 1269
DragLeave event 252
DragObject functions
  ClassName 429
  Drag 543
  Hide 778
  Move 959
  PointerX 1033
  PointerY 1034
  PostEvent 1049
  Print 1058
  Resize 1142
  SetFocus 1276
  SetPosition 1303
  SetRedraw 1308
  Show 1369
  TriggerEvent 1444
  TypeOf 1450
DragWithin event 254
Draw function 546
drawing objects
  and SetFocus function 1276
  posting events 1049
  setting color of 1151
DrawObject functions
  ClassName 429
  Hide 778
  Move 959
  Print 1058
  Resize 1142
  Show 1369
  TypeOf 1450
DropDownListBox control
  deleting text 432
  deleting values 434
  obtaining values of 771
DropDownListBox functions
  AddItem 394
  Clear 432
  Copy 465
  Cut 478
  DeleteItem 510
  DirList 531
  DirSelect 533
  DraggedObject 545
  FindItem 595
  InsertItem 818
  Paste 1025
  Position 1041
  Post 1047
  ReplaceText 1128
  Reset 1132
  SelectedLength 1204
  SelectedStart 1209
  SelectedText 1211
  SelectItem 1213
  SelectText 1220
  Text 1417
  TotalItems 1428
DropDownPictureListBox functions
  AddItem 395
  AddPicture 401
  Clear 432
  Copy 465
  Cut 478
  DeletePicture 516
  DeletePictures 517
  FindItem 595
  InsertItem 820
  Paste 1025
  Position 1041
  ReplaceText 1128
  SelectedLength 1204
  SelectedStart 1209
  SelectedText 1211
  SelectItem 1213
  SelectText 1220
  Text 1417
  TotalItems 1428
dwItemStatus enumerated data type 705
DWObjects, OLE functions 385, 465, 541, 1460
dynamic libraries 1291
dynamic library (DLL) 1393
dynamic SQL
  about 179
  considerations 181
  DynamicDescriptionArea 180
  DynamicStagingArea 180
  Format 1 183

Format 2 184
Format 3 186
Format 4 189
formats listed 179
NULL values 184, 187
ordering statements 181
preparing DynamicStagingArea 181
statements 180
dynamic SQL functions
GetDynamicDate 669
GetDynamicDateTime 671
GetDynamicNumber 673
GetDynamicString 674
GetDynamicTime 675
SetDynamicParm 1270
DynamicDescriptionArea
about 180
properties 190
DynamicStagingArea
about 180
preparing 181

E

edit control
applying contents of 383
counting lines in 883
DataWindow 383
deleting text from 432
determining insertion point position 1041
inserting clipboard contents 435
obtaining value in 756
replacing text 1128
selected text 1204, 1209
setting value of 1333
edit style 771
EditChanged event 258
EditLabel function 548
EditMask functions
CanUndo 421
Clear 432
Copy 465
Cut 478
GetData 652
LineCount 883
LineLength 885
Paste 1025
Position 1041
ReplaceText 1128
Scroll 1182
SelectedLength 1204
SelectedLine 1206
SelectedStart 1209
SelectedText 1211
SelectText 1220
SetMask 1293
TextLine 1418
Undo 1454
embedded SQL 153
Enable function 550
Enabled property 778, 1308
EndLabelEdit event 259
EntryList function 551
enumerated data types 34
envelope 906
environment
getting information about 676
TEMP variable 907
error checking
cascaded calls 116
compiler 106
Error DataWindow 1124
Error event 261
error handling
after SQL statements 156
calling functions or events 109
error objects, creating 135
errors
calling functions or events 107
displaying pipeline 1386
during execution 76
reporting on database 497, 499
update 766
escape sequences 1058, 1087
Evaluate function 526
EventParmDouble 553
EventParmString 554
events
about 98, 196
adding to queue 1049
ancestor 121

- and hidden objects 778
- and print jobs 1071
- arguments 112, 196
- cascaded calls 115, 118
- defined 98
- errors when calling 107
- extending 111
- finding 101
- for DataWindow printing 1058
- IDs, with and without 196
- overriding 111
- posting 102, 116, 1414
- return codes 197
- return values 115, 197
- similarities to functions 99
- static and dynamic 103
- system 98, 196
- triggering 102, 196, 1415, 1444
- types of 196
- user-defined 196, 199

Excel file 1168
exclamation point icon 930
exclusive share mode 978, 981, 982
ExecRemote function 555
executable
- returning application handle 775
- running 1164

EXECUTE statement 168, 1270
execution errors 106
EXIT statement 143
Exp function 559
ExpandAll function 560
ExpandItem function 561
exponent 559
exponentiation operator 74
expressions
- Any data type 30
- checking for NULL 856
- data type promotion 80
- data types 80
- DataWindows and Any data type 31
- evaluating 524
- for Modify function 942
- in declaration 44
- literals 81
- operators and data types 81

external functions 61
ExternalException event 264

F

Fact function 562
FETCH statement 169
file functions
- FileClose 563
- FileDelete 564
- FileExists 565
- FileLength 566
- FileOpen 568
- FileRead 571
- FileSeek 574
- FileWrite 575
- GetFileOpenName 677
- GetFileSaveName 679

FileClose function 563
FileDelete function 564
FileExists event 267
FileExists function 565
FileLength function 566
FileOpen function 568
FileRead function 571
files
- importing data from 788
- linking 887
- security and sharing violation 566

FileSeek function 574
FileWrite function 575
Fill function
- about 577
- and printing 577

FillPattern 663, 1264, 1321
filter buffer
- modified rows 939
- resetting update flags 1140
- retrieving data from 694, 697, 699, 702, 707, 709
- returning modified rows 721
- sharing data 1360, 1363

Filter function 578
FilteredCount function 580
filters
- applying 1146

filenames 677, 679
setting criteria 1272
Find function 582
FindCategory function 588
FindClassDefinition function 590
FindFunctionDefinition function 592
FindGroupChange function 593
FindItem function 595
FindMatchingFunction function 602
FindNext function 605
FindRequired function 606
FindSeries function 610
FindTypeDefinition function 612
flags, update 1140
flicker 1308
focus
and line length 885
column 638, 641
finding control with 684
selected text 1204, 1209, 1211, 1220
setting 1276, 1315
folder 877
fonts
and string length when printing 1094
defining for printing 1073
FontFamily enumerated data type 1073
FontPitch enumerated data type 1073
names and sizes 1074
setting 1089
when printing 1061
when printing DataWindow controls 1072
footer
locating 625
moving objects to 1305
foreground color
data points 660, 1261
series 741, 1317
foreground layer of DataWindow 1305
Form presentation style 1410
formats
applying to strings 1404
of columns 685, 1277
of filter criteria 1272
sort criteria 1325
formfeed, specifying 9
frame window 1035, 1469, 1470
function object
exporting as syntax 879
listing 877
recreating from syntax 881
functions
about 98
access level for external 63
alphabetical list of 381
ancestor 121
arguments 112
calling global 118
calling system 118
cascaded calls 115, 118
case sensitivity 119
chars as arguments 83
creating external 69
DLLs 63
errors when calling 107
external 61
external data types 66
external, defined 99
external, mail 902
external, obtaining handles of objects 1047
external, reporting database handle 501
finding 101
overloading 110
overriding 110
posting 102, 116
return values 114
similarities to events 99
static and dynamic 103
system, defined 99
triggering 102
type promotion 110
user-defined 99

G

garbage collection 136, 139
GarbageCollect function 614
GarbageCollectGetTimeLimit function 615
GarbageCollectSetTimeLimit function 616
GenerateHTMLForm function 617
GetActiveSheet function 619
GetAlignment function 620

GetApplication function 621
GetArgElement function 622
GetAutomationNativePointer function 623
GetBandAtPointer function 625
GetBorderStyle function 627
GetChanges function 628
GetChild function 631
GetChildrenList function 634
GetClickedColumn function 636
GetClickedRow function 637
GetColumn function 638
GetColumnName function 641
GetCommandDDE function 642
GetCommandDDEOrigin function 644
GetCompanyName function 645
GetContextKeywords function 646
GetContextService function 648
GetData function 650
GetDataDDE function 655
GetDataDDEOrigin function 656
GetDataPieExplode function 658
GetDataStyle function 660
GetDataValue function 667
GetDynamicDate 190
GetDynamicDate function 669
GetDynamicDateTime 190
GetDynamicDateTime function 671
GetDynamicNumber 190
GetDynamicNumber function 673
GetDynamicString 190
GetDynamicString function 674
GetDynamicTime 190
GetDynamicTime function 675
GetEnvironment function 676
GetFileOpenName function 677
GetFileSaveName function 679
GetFirstSheet function 681
GetFixesVersion function 682
GetFocus event 268
GetFocus function 684
GetFormat function 685
GetFullState function 686
GetHostObject function 688
GetItem function 690
GetItemDate function 694
GetItemDateTime function 697
GetItemDecimal function 699
GetItemNumber function 702
GetItemStatus function 705
GetItemString function 707
GetItemTime function 709
GetLastReturn function 712
GetMajorVersion function 714
GetMessageText function 716
GetMinorVersion function 717
GetName function 719
GetNativePointer function 720
GetNextModified 721
GetNextSheet function 723
GetObjectAtPointer function 725
GetOrigin function 726
GetParagraphSetting function 727
GetParent function 728
GetRemote function 730
GetRow function 734
GetRowFromRowId function 735
GetRowIdFromRow function 737
GetSelectedRow function 739
GetSeriesStyle function 740
GetServerInfo function 747
GetShortName function 749
GetSQLPreview function 751
GetSQLSelect function 752
GetStateStatus function 754
GetText function 756
GetToolbar function 759
GetToolbarPos function 761, 1340
GetTrans function 764
GetUpdateStatus function 766
GetURL function 769
GetValidate function 770
GetValue function 771
GetVersionName function 773
global functions
  calling 118
  defined 99
global scope operator 38
global transaction objects 1350
global variables
  about 36, 37
  scope operator 38
GOTO statement 146

Graph functions
- AddCategory 387
- AddData 391
- AddSeries 403
- CategoryCount 422
- CategoryName 423
- Clipboard 436
- DataCount 480
- DeleteCategory 504
- DeleteData 507
- DeleteSeries 519
- FindCategory 588
- FindSeries 610
- GetData 650
- GetDataPieExplode 658
- GetDataStyle 660
- GetSeriesStyle 740
- ImportClipboard 785
- ImportFile 790
- ImportString 795
- InsertCategory 808
- InsertData 812
- InsertSeries 837
- ModifyData 955
- Reset 1132
- SaveAs 1170
- SeriesCount 1231
- SeriesName 1232
- SetDataPieExplode 1258
- SetDataStyle 1260
- SetSeriesStyle 1317

graphics
- printing 1066
- properties of 524
- under pointer 725

graphs
- categories 392
- overlay 745
- series 403

grColorType enumerated data type 660
grDataType enumerated data type 651, 667
Grid presentation style 1410
grObjectType enumerated data type 966
Group presentation style 1410
GroupCalc function 774
groups
- filtering 578
- recalculating levels 774
- sorting 1378

grResetType enumerated data type 1133
grSymbolType enumerated data type 1322

H

HALT statement 147
handle
- application 625
- database 501
- DDE 442, 987, 1395
- mailSession object 902, 1229
- validating 862

Handle function 775
header band
- locating 625
- moving objects to 1305

Hebrew functions
- IsAllHebrew 848
- IsAnyHebrew 850
- IsHebrew 854
- IsHebrewAndNumbers 855

height
- object 1142
- workspace 1466

Help
- calling Winhelp 1371
- displaying MicroHelp 1295

Help Search window 1371
hidden objects 1369
Hide event 270
Hide function 778
hierarchies
- child items in a list 825, 828, 831
- items in TreeView 451, 561
- sorting 1381
- sorting children 1378
- system 32, 429

high word of long 843
highlighting
- items in lists 1213, 1397
- rows 859, 1218
- scrolling 1186, 1189, 1191, 1194

setting 1331
horizontal fill pattern 1265, 1322
horizontal scrollbar for lists 394
horizontal scrolling, when adding items to lists 394
host variables 155
hot link
about 1256
determining origin of 656
determining source of data 656
establishing 1393
terminating 1400
HotLinkAlarm event 271
Hour function 780
hourglass pointer 1301
HyperlinkToURL function 781
hyphens, prohibiting in variable names 6

I

icons
arranging in ListView 407
arranging windows 408
in message box 930
identifier names, rules for 6
Idle event 272
IDs for events 196
IF...THEN statement
about 148
multiline 149
single-line 148
image
assigning to picture control 1300
retrieving from clipboard 435
setting row focus indicator 1315
ImportClipboard function 784
ImportFile function 788
importing, data 788, 793
ImportString function 793
inbox
deleting messages from 898
downloading messages to 905
reading mail messages 906
retrieving message IDs from 898, 900
saving messages in 914
IncomingCallList function 798
index
highlight state of 1331, 1397
obtaining top 1426
of listbox item 1202, 1214
indicator variables 155
Inet objects
GetURL function 769
HyperlinkToURL function 781
PostURL function 1053
Information icon 930
inheritance 91
back quote 131
double colon 131
PowerBuilder objects 32
INI file
reading 1097, 1099
writing values to 1306
input fields in rich text 800, 802, 803, 804, 805, 806
InputFieldChangeData function 800
InputFieldCurrentName function 802
InputFieldDeleteCurrent function 803
InputFieldGetData function 804
InputFieldInsert function 805
InputFieldLocate function 806
InputFieldSelected event 273
Insert Object dialog 834
INSERT statement 170
InsertCategory function 808
InsertClass function 810
InsertColumn function 811
InsertData function 812
InsertFile function 817
inserting strings 1126
insertion point
character position 1201
in editable controls 885
in text line 1206, 1418
when pasting from clipboard 1025
InsertItem event 274
InsertItem function 818
InsertItemFirst function 825
InsertItemLast function 828
InsertItemSort function 831
InsertObject function 834
InsertPicture function 835
InsertRow function 836

InsertSeries function 837
instance variables
- about 36, 37
- class of 429
- dot notation 38
- initialized 45

instances
- checking if valid 862
- defined 88
- of user object 996, 1000, 1005, 1009

Int function 839
integer
- combining into long value 892
- converting to 840
- converting to char 426
- obtaining from blob 840

integer data type 25
Integer function 840
Intel 676
internal transaction object 1139, 1348
InternetData function 842
InternetRequest objects
- InternetData function 842

interpersonal messages 900
interprocess messages 900
interval 1422
IntHigh function 843
IntLow function 844
InvokePBFunction function 845
IsAllArabic function 847
IsAllHebrew function 848
IsAnyArabic function 849
IsAnyHebrew function 850
IsArabic function 851
IsArabicAndNumbers function 852
IsDate function 853
IsHebrew function 854
IsHebrewAndNumbers function 855
IsNull function 856
IsNumber function 840, 857
IsPreview function 858
IsSelected function 859
IsTime function 861
IsValid function
- about 862
- and Handle function 775
- description 862
- getting active sheet 619
- getting open sheets 681, 723

ItemChanged event 275, 383, 528, 756, 1458
ItemChanging event 278
ItemCollapsed event 279
ItemCollapsing event 280
ItemError event 281, 383, 756
ItemExpanded event 284
ItemExpanding event 285
ItemFocusChanged event 286
ItemPopulate event 288
items
- adding to lists 394, 818
- deleting from list 510, 1132
- determining number of selected 1429
- determining total number of 1428
- editing 756
- highlight state of 1331, 1397
- index number of 1202
- linking 887
- selecting 1213
- setting value of 1358
- text of 1203, 1417
- top 1345, 1426

K

Key event 289
keyboard
- determining key pressed 862
- selecting text 466

KeyCode enumerated data type
- about 862
- values 863

KeyDown function 863

L

Label presentation style 1410
label, under pointer 725
labels for GOTO 8
language for OLE automation 1239, 1243
Layer enumerated data type 408

Layered window 994
layering opened windows 990
layout 1072
LeftTrim function 868
Len function 869
length
  line 885
  OLE stream 871
  selected text 1204
  string or blob 869
Length function 871
LibDirType enumerated data type 877
LibExportType enumerated data type 879
libraries
  deleting objects from 877
  pasting and linking object from 1027
  search path 1291
Library functions
  LibraryCreate 873
  LibraryDelete 875
  LibraryDirectory 877
  LibraryExport 879
  LibraryImport 881
LibraryCreate function 873
LibraryDelete function 875
LibraryDirectory function 877
LibraryExport function 879
LibraryImport function 881
limit, numeric 424
line spacing
  setting 1090
  when printing text 1061
LineCount function 883
LineDown event 291
LineLeft event 292
LineLength function 885
LineList function 886
LineRight event 293
lines
  and SetFocus function 1276
  color for data points 660
  counting number of 883
  deleting and adding 941
  determining length 885
  graphs, color for data points 1261
  graphs, color for series 741, 1317
  graphs, style for data points 663, 1262
  graphs, style for series 742, 743, 1319
  printing 1075, 1092
  scrolling 1182
  selected text 1206
  spacing in rich text 750
  text 1418
  under pointer 725
  width 663
LineUp event 294
linking
  clipboard contents 1027, 1030
  establishing 887
LinkTo function 887
ListBox functions
  AddItem 394
  DeleteItem 510
  DirList 531
  DirSelect 533
  FindItem 595
  InsertItem 818
  Reset 1132
  SelectedIndex 1202
  SelectedItem 1203
  SelectItem 1213
  SetState 1331
  SetTop 1345
  State 1397
  Text 1417
  Top 1426
  TotalItems 1428
  TotalSelected 1429
Listen function 889
lists
  adding items 818
  adding new item 394
  deleting items from 1132
  horizontal scrollbar 394
  of files in listbox 531
  of objects in libraries 877
  sorted 395, 396
ListView control, columns 1283
ListView functions
  AddColumn 389
  AddItem 397, 398
  AddLargePicture 400

AddSmallPicture 405
AddStatePicture 406
Arrange 407
DeleteColumn 505
DeleteColumns 506
DeleteItem 511
DeleteLargePicture 514
DeleteLargePictures 515
DeleteSmallPicture 520
DeleteSmallPictures 521
DeleteStatePicture 522
DeleteStatePictures 523
EditLabel 548
FindItem 596, 598
GetColumn 639
GetItem 690
GetOrigin 726
InsertColumn 811
InsertItem 821
ListView 1427
SelectedIndex 1202
SetItem 1282
SetOverlayPicture 1297
Sort 1379
TotalItems 1428
TotalSelected 1429
literals
data types of 81
specifying 24, 25, 26, 28
local variables 36, 37
locks 1348
Log function
about 890
inverse 890
logarithms 890, 891
logical operators 76
LogTen function
about 891
inverse 891
long data type
about 26
converting to 892
returning high word 843
returning low word 844
Long function 892
LongParm
posting events 1049
specifying values for 892
triggering events 1444
loops
about 140
avoiding infinite 1250, 1313, 1457
iterative 144
leaving 143
skipping current iteration 134
yielding within 1474
LoseFocus event 295, 383, 931
Lotus 1-2-3 format 1168
low word of long 844
Lower function 894
LowerBound function 895
lowercase 894

M

Macintosh
AppleScript script 538
defining fonts for printing 1074
displaying Save File response window 679
DoScript function 538
getting filenames 677
getting information about 676
Handle function 775
handles of external objects 1047
initialization files 1099
mail functions, no effect on 896, 904
OLE functions, no effect on 968
unavailable DDE functions 1143, 1256, 1310
mail functions
mailAddress 896
mailDeleteMessage 898
mailGetMessages 900
mailHandle 902
mailLogoff 903
mailLogon 904
mailReadMessage 906
mailRecipientDetails 909
mailResolveRecipient 911
mailReturnCode 904
mailSaveMessage 914
mailSend 917

mailAddress function 896
mailDeleteMessage function 898
mailHandle function 902
mailLogoff function 903
mailLogon function 904
mailLogonOption enumerated data type 904
mailReadMessage function 906
mailReadOption enumerated data type 907
mailRecipient structure 911
mailRecipientDetails function 909
mailResolveRecipient function 911
mailReturnCode function 904
mailSaveMessage function 914
mailSend function 917
main window 959
MAPI 902
margins 1061, 1087, 1299
masks
  applying to strings 1404
  matching 919
  reporting length of 885
  setting 1293
Match function 919
Max function 923
maximum value below a limit 839
maximum value of two numbers 923
MDI Client (MDI_1) functions
  ClassName 429
  Hide 778
  Print 1058
  Resize 1142
  SetRedraw 1308
  Show 1369
  TypeOf 1450
MDI frame
  arranging windows 408
  changing menus 425
  displaying popup menus 1035
  getting active 619
  opening sheets 972, 990, 993
  specifying MicroHelp text 1295
MDI frame functions
  ArrangeSheets 408
  GetActiveSheet 619
  GetFirstSheet 681
  GetNextSheet 723
  GetToolbar 759
  GetToolbarPos 761, 1340
  OpenSheet 990
  OpenSheetWithParm 993
  Print 1058
  SetMicroHelp 1295
  SetToolbar 1338
measurement 1455
member, OLE 924, 926, 928
MemberDelete function 924
MemberExists function 926
MemberRename function 928
memory
  allocation for arrays 55
  and variable-sized arrays 1463
  managing 91
  releasing after mail session 903
Menu functions
  Check 427
  ClassName 429
  Disable 535
  Enable 550
  PopMenu 1035
  Show 1369
  TriggerEvent 1444
  TypeOf 1450
  Uncheck 1452
Menu objects
  exporting as syntax 879
  listing 877
  recreating from syntax 881
menus
  changing 425
  Checked property 427
  creating object 135
  displaying 1035
  for sheet 990
message ID array 900
Message object
  accessing parameters 1014
  and TriggerEvent function 1444
  close return value 448
  creating 135
  determining type 1452
  extracting strings from 1405, 1408
  open sheet parameters 993

PowerObjectParm property 448
properties 1001, 1003, 1010, 1012
specifying values for 892
MessageBox function 930, 1077
messages
database error 499
deleting 898
posting 1047
retrieving text 716
saving 914, 917
sending to a window 1229
metacharacters 919
MicroHelp 1295
Microsoft Multiplan format 1168
Microsoft Windows
and DDE 730
and timers 1422
calling Winhelp 1371
defining fonts for printing 1074
displaying Save File response window 679
events and messages in 1051
getting filenames 677
getting information about 676
message numbers 1229
obtaining handle 775
returned messages 843, 844
RightToLeft version 847, 848, 849, 850, 851, 852, 854, 855, 1150
Mid function 933
Min function 936
minimum value
above a limit 424
of two numbers 936
Minute function 937
miscellaneous functions
IsValid 862
KeyDown 863
MessageBox 1032
PixelsToUnits 1032
RGB 1151
SetNull 1296
SetPointer 1301
TypeOf 1450
UnitsToPixels 1455
Mod function 938
Modified event 297
ModifiedCount function 939
Modify function 941
ModifyData function 955
modulus 938
monitor 676
Month function 958
month, obtaining the day of 489
More Windows menu item 991
mouse
selecting text 466
setting shape of pointer 1301
MouseDown event 299
MouseMove event 302
MouseUp event 306
Move function 959
Moved event 309
multidimensional arrays 53, 57
MultiLineEdit functions
CanUndo 421
Clear 432
Copy 465
Cut 478
LineCount 883
LineLength 885
Paste 1025
Position 1041
ReplaceText 1128
Scroll 1182
SelectedLength 1204
SelectedLine 1206
SelectedStart 1209
SelectedText 1211
SelectText 1220
TextLine 1418
Undo 1454
multiplication 74, 75
MultiSelect property
highlighted state 1331, 1400
selecting items 1202, 1203, 1215

N

names, rules for 6
naming conventions 42
negative numbers 1373

nested OLE objects 979, 982
New item status
  resetting 1140
  setting 1287
newline, specifying 9
NewModified item status
  resetting 1140
  returning next row with 721
  setting 1287
NextActivity function 962
NoBorder border style 627
NOT operator 76
NotModified item status
  resetting 1140
  setting 1287
Now function 964
null object references 994, 1001, 1003, 1010, 1012, 1015, 1017
NULL values
  about 11
  checking 856
  dynamic SQL 187
  in boolean expressions 77
  in sort criteria format 1325
  setting variables to 1296
  testing for 11
numbers
  category 423
  checking string 857
  comparing 923, 936
  converting char 426, 484, 503
  determining maximum 424
  determining sign of 1373
  getting dynamic 673
  logarithm of 890, 891
  multiplying by pi 1031
  of day of week 491
  of lines, counting 883
  of rows in buffers 766
  random 1101, 1102
  retrieving from buffers 699, 702
  returning remainder 938
  rounding 1155
  truncating 1449
numeric functions
  Abs 382
  Ceiling 424
  Cos 469
  Exp 559
  Fact 562
  Int 839
  Log 890
  Max 923
  Min 936
  Mod 938
  Pi 1031
  Rand 1101
  Randomize 1102
  Round 1155
  Sign 1373
  Sin 1376
  Sqrt 1384
  Tan 1416
  Truncate 1449
N-Up presentation style 1410

O

ObjectAtPointer function 965
objects
  about 88
  ancestor 91
  assignment 93
  changing position 1305
  creating instance 135
  deleting and adding 953
  deleting from libraries 875
  destroying instance 139
  determining class of 429
  determining type 1450
  garbage collection 139
  general references 14
  hiding 778, 959
  inserting 810, 817, 834
  instantiating 90
  linking 887
  loading 1291
  memory 91
  moving 959
  naming 525
  obtaining handle 775

parent object 728
passing as arguments 112
posting events 1049
recreating 881
redrawing 1308
reference, defined 88
saving OLE 1166
selecting 1217
setting focus 1276
specifying as a column 525
triggering events 1444
under pointer 725, 965
objects, Connection
ConnectToServer function 464
CreateInstance function 474
DisconnectServer function 537
GetServerInfo function 747
RemoteStopConnection function 1121
RemoteStopListening function 1123
objects, ConnectionInfo
GetServerInfo function 747
objects, remote
SetConnect function 1252
objects, shared
SharedObjectDirectory function 1364
SharedObjectGet function 1365
SharedObjectRegister function 1367
SharedObjectUnregister function 1368
objects, Transport
Listen function 889
StopListening function 1402
Offsite enumerated data type 385
OK button 930
OLE DWObject functions
Activate 385
Copy 465
DoVerb 541
UpdateLinksDialog 1460
OLE expressions and Any data type 31
OLEActivate 968
OLEControl functions
Activate 385
Clear 432
Copy 465
Cut 478
DoVerb 541
GetData 653
GetNativePointer 720
InsertClass 810
InsertFile 817
InsertObject 834
LinkTo 887
Open 970
Paste 1025
PasteLink 1027
PasteSpecial 1030
ReleaseAutomationPointer 1120
Save 1166
SaveAs 1172, 1173
SelectObject 1217
SetAutomationLocale 1239
SetData 1254
UpdateLinksDialog 1460
OLECustomControl functions
GetData 653
GetNativePointer 720
ReleaseAutomationPointer 1120
SetAutomationLocale 1239
SetData 1254
OLEObject functions
ConnectNewToObject 454
ConnectToNewRemoteObject 456
ConnectToObject 458
ConnectToRemoteObject 461
DisconnectObject 536
GetAutomationNativePointer 623
ReleaseAutomationPointer 1119
SetAutomationPointer 1241
SetAutomationTimeout 1243
OLEStorage functions
Clear 432
Close 439
MemberDelete 924
MemberExists 926
MemberRename 928
Open 970
SaveAs 1175, 1176
OLEStream functions
Close 440
Length 871
Open 970
Read 1103

Seek 1199
Write 1471
OPEN Cursor statement 171
Open event 310, 1145
Open function 970
OpenChannel function 987
OpenSheet function 990
OpenSheetWithParm 993
OpenTab function 996
OpenTabWithParm function 1000
OpenUserObject function 1005
OpenUserObjectWithParm function 1009
OpenWithParm 1014
operating system
information about 676
RightToLeft version 847, 848, 849, 850, 851, 852, 854, 855, 1150
operators
about 74
arithmetic 74
assignment shortcuts 128, 129
concatenation 78
effect on data types 81
logical 76
precedence 79
relational 76
OR operator 76
ORACLE 950
Original window 994
Other event 313
OutgoingCallList function 1019
oval
and SetFocus function 1276
printing 1079
overflow on assignment 81
overlay 745, 1323

P

page
printing 1081
printing borders 1079, 1082, 1084
size 1061
PageCreated function 1022
PageDown event 314
PageLeft event 316
PageRight event 317
PageUp event 318
paging functions
ScrollNextPage 1183
ScrollPriorPage 1189
paragraphs 1299
parameters
command line 452
opening sheets with 993
opening tab pages with 1000
opening user objects with 997, 998, 1006, 1008, 1009
opening windows with 1014
setting in transaction object 1349, 1350
specifying for DynamicDescriptionArea 1270
Parent pronoun 15
parent window
changing position relative to 959
obtaining 1023
of open window 971, 972, 1014
parentheses in expressions 79
ParentWindow function 1023
parsing strings 1039
password 905
Paste function 1025
PasteLink function 1027
PasteSpecial function 1030
pasting
embedding or linking 1030
from clipboard 1025, 1027
path
of library file 873
OLE storage 972
returning 677
saving files 679
pattern matching 919
PBL file
creating 873
deleting 875
listing contents of 877
pbm_dwngraphcreate event 1318
PBSELECT statement 525, 752
PDB file 974
performance
and SetTrans function 1348

- and SetTransObject function 1350
- and transaction objects 1140
- and Yield function 1474
- Any data type 31
- dynamic function and event calls 106

period in text patterns 919
Pi function 1031
Picture control 1315
Picture functions
- ClassName 429
- Drag 543
- Draw 546
- Hide 778
- Move 959
- PointerX 1033
- PointerY 1034
- PostEvent 1049
- Print 1058
- SetFocus 1276
- SetPicture 1300
- SetPosition 1303
- SetRedraw 1308
- Show 1369
- TriggerEvent 1444
- TypeOf 1450

PictureListBox functions
- AddItem 395
- AddPicture 401
- DeletePicture 516
- DeletePictures 517
- FindItem 595
- InsertItem 820
- SelectedItem 1203
- SelectItem 1213
- SetTop 1345
- State 1397
- Text 1417
- Top 1426
- TotalItems 1428
- TotalSelected 1429

pictures
- as row focus indicators 1316
- for TreeView items 1289
- in list boxes 401
- in rich text 835
- in TreeView controls 401
- ListView controls 400, 405, 406
- overlay in lists 1297
- TreeView controls 406

PictureSelected event 320
pie graphs 658, 1258
PIF file 1164
PipeEnd event 321
Pipeline functions
- Cancel 420
- Repair 1124
- Start 1385

PipeMeter event 322
PipeStart event 323
pixels 1032, 1455
PixelsToUnits function 1032
plus sign in text patterns 920
point size 1073
pointer
- determining distance from edge 1033
- distance from top 1034
- file 574, 575
- locating bands 625
- read/write 1199
- returning object under 725, 965
- setting shape 1301

PointerX function 1033
PointerY function 1034
pointing hand 1315
polymorphism for functions and events 103
PopMenu function 1035
PopulateError function 1037
popup windows
- moving 959
- obtaining parent 1023
- opening 972, 1014

Pos function 1039
position
- changing 959
- of insertion point 1041
- setting for control 1303

Position function 1041
positive numbers 1373
Post function 1047
PostEvent function 1049
posting
- functions or events 102

- restrictions 102
- PostURL function 1053
- PowerBuilder
  - data types for external functions 66
  - Unicode and ANSI versions 1424, 1430
- PowerBuilder units 1032, 1455
- PowerObject base class 32, 88
- PowerObject functions
  - ClassName 429
  - GetContextService 648
  - GetParent 728
- PowerObjectParm
  - and CloseWithReturn function 448
  - determining type 1452
  - opening sheets with parameters 994, 1001, 1003, 1010, 1012
- PowerScript statements 128
- precedence, operator 79
- presentation style 1410
- primary buffer
  - modified rows 939
  - resetting update flags 1140
  - restoring rows to 1273
  - retrieving data from 694, 697, 699, 702, 707, 709
  - returning modified rows 721
  - row count 1157
  - sharing data 1360, 1363
- primary DataWindow control 1360, 1361, 1363
- print cursor
  - getting coordinates of 1095, 1096
  - in print jobs 1061
- Print function 1057
- print functions
  - Print 1057
  - PrintBitmap 1066
  - PrintCancel 1068
  - PrintClose 1071
  - PrintDataWindow 1072
  - PrintDefineFont 1073
  - PrintOpen 1077
  - PrintOval 1079
  - PrintPage 1081
  - PrintRect 1082
  - PrintRoundRect 1084
  - PrintScreen 1086
  - PrintSend 1087
  - PrintSetFont 1089
  - PrintSetSpacing 1090
  - PrintSetup 1091
  - PrintText 1092
  - PrintWidth 1094
  - PrintX 1095
  - PrintY 1096
- print job 1077
- PrintBitmap function 1066
- PrintCancel function 1068
- PrintClose function 1071
- PrintDataWindow function 1072
- PrintDefineFont function 1073
- PrintEnd event 324
- printer setup 1087
- Printer Setup dialog box 1091
- PrintFooter event 325
- PrintHeader event 327
- PrintLine function 1075
- PrintOpen function
  - about 1077
  - and message boxes 931
- PrintOval function 1079
- PrintPage event 329
- PrintPage function 1081
- PrintPreview display 941
- PrintRect function 1082
- PrintRoundRect function 1084
- PrintScreen function 1086
- PrintSend function 1087
- PrintSetFont function 1089
- PrintSetSpacing function 1090
- PrintSetup function 1091
- PrintStart event 330
- PrintText function 1092
- PrintWidth function 1094
- PrintX function 1095
- PrintY function 1096
- private access
  - functions 63
  - variables and constants 46
- PRIVATEREAD access modifier 46
- PRIVATEWRITE access modifier 46
- processor 676
- profile files
  - reading 1097, 1099

writing to 1306
ProfileClass objects
RoutineList function 1156
ProfileInt function 1097
ProfileLine objects
OutgoingCallList function 1019
ProfileRoutine objects
IncomingCallList function 798
LineList function 886
OutgoingCallList function 1019
ProfileString function 1099
Profiling functions
BuildModel 417
ClassList 428
DestroyModel 530
RoutineList 1156
SetTraceFileName 1346
SystemRoutine 1413
Prompt For Criteria 941, 950
pronouns
about 14
instance variables 39
Parent 15
Super 17
This 16
properties
and GetFocus function 684
DataWindow 944
font, for printing 1073
getting and setting 621
Message object 994
reporting values of 524
setting width and height 1142
syntax 525
window 971, 973
property expressions, Any data type 31
PropertyChanged event 331
PropertyRequestEdit event 332
protected access
functions 63
variables and constants 46
PROTECTEDREAD access modifier 46
PROTECTEDWRITE access modifier 46
public access
functions 63
variables and constants 46

Q

Query mode 941, 950
question mark
dynamic SQL 184, 186, 189
icon in message box 930
in text patterns 920
quoted strings, continuing 19
quotes
in Modify function 943, 950
in property values 525
in sort criteria 1325
nesting 27
rules for 28
specifying 9
with tilde 27

R

radians 1031
RadioButton edit style 771
Rand function 1101
random numbers
initializing generator 1102
obtaining 1101
Randomize function 1102
RButtonDown event 333
RButtonUp event 336
Read function 1103
read-only arguments 112
real data type 26
Real function 1106
recipient, mail 909
rectangle
and SetFocus function 1276
printing 1082, 1084
setting row focus indicator 1315
recursive call 1250
reference, by 112
references
and CloseWithReturn function 449
passing parameters 994, 1001, 1003, 1010, 1012, 1015, 1017
to child window 631
RegEdit utility 968
Registration database 812

RegistryDelete function 1108
RegistryGet function 1109
RegistryKeys function 1111
RegistrySet function 1113
RegistryValues function 1116
relational operators 76
RelativeDate function 1117
RelativeTime function 1118
ReleaseAutomationNativePointer function 1119
ReleaseNativePointer function 1120
remainder 938
remote access 1348
remote DDE application 1143
remote objects, SetConnect function 1252
remote procedure calls
  declaring 70
  defined 99
RemoteExec event 337, 642, 1395
RemoteHotLink event 338
RemoteHotLinkStart event 1395
RemoteHotLinkStop event 339, 1395
RemoteRequest event 340, 1256, 1395
RemoteSend event 341, 656, 1395
RemoteStopConnection function 1121
RemoteStopListening function 1123
Rename event 342
Repair function 1124
repairing pipeline, canceling 420
Replace function 1126
ReplaceText function 1128
report view for ListView 690
reports, nested 631
ReselectRow function 1130
reserved words 13
reset flag argument 1457
Reset function 1131
ResetArgElements function 1135
ResetDataColors function 1137
ResetTransObject function 1139
ResetUpdate function 1140
Resize event 343
Resize function 1142
resource files 1393
RespondRemote function 1143
response windows
  closing 448
  moving 959
  running applications from 1165
Restart function 1145
Retrieve function 1146
Retrieve Only As Needed 941, 952
RETRIEVE statement 1350
RetrieveEnd event 344
RetrieveRow event 345, 494
RetrieveStart event 346, 1146
retry button 930
return count 1146
RETURN statement 151
return values
  about 114
  event return codes 197
  from ancestor events 121, 197
  from mail session 904
  SQL 1350
  TriggerEvent function 1444
Reverse function 1150
RGB function 1151
rich text
  alignment 620, 1236
  and data 481
  copying with formatting 467, 1029
  data 800, 802, 803, 804, 805, 806
  determining insertion point position 1042
  editing header and footer 1370
  find again 605
  finding text 586
  formatting 727, 1299
  line spacing 1327
  preview 858
  preview document 858, 1055
  printing 1063
  save file 1180
  selecting 1222
  selecting a line 1226
  selecting a word 1227
  selecting all 1225
  text color 757, 1335
  text settings 1336
rich text formatting 750, 758
RichTextEdit functions
  CanUndo 421
  Clear 432

Copy 465
CopyRTF 467
Cut 478
DataSource 481
Find 586
FindNext 605
GetAlignment 620
GetParagraphSetting 727
GetSpacing 750
GetTextColor 757
GetTextStyle 758
InputFieldChangeData 800
InputFieldCurrentName 802
InputFieldDeleteCurrent 803
InputFieldGetData 804
InputFieldInsert 805
InputFieldLocate 806
InsertPicture 835
IsPreview 858
LineCount 883
LineLength 885
Paste 1025
PasteRTF 1029
Position 1042
Preview 1055
Print 1063
ReplaceText 1128
SaveDocument 1180
Scroll 1182
ScrollNextPage 1184, 1187
ScrollPriorPage 1190
ScrollPriorRow 1192
ScrollToRow 1195
SelectedColumn 1201
SelectedLength 1204
SelectedLine 1206
SelectedPage 1208
SelectedStart 1209
SelectedText 1211
SelectText 1222
SelectTextAll 1225
SelectTextLine 1226
SelectTextWord 1227
SetAlignment 1236
SetParagraphSetting 1299
SetSpacing 1327
SetTextColor 1335
SetTextStyle 1336
ShowHeadFoot 1370
Undo 1454
Right function 1153
RightClicked event 348
RightDoubleClicked event 350
RightToLeft operating system 1150
RightToLeft software 847, 848, 849, 850, 851, 852, 854, 855
RightTrim function 1154
ROLLBACK statement 172
Round function 1155
RoutineList function 1156
RowCount function 1157
RowFocusChanged event 352
RowFocusChanging event 353
rows
canceling retrieval 494
clicked 637
copying 1159
correcting pipeline data 1124
deleting 508, 518
determining insertion point position 1041
displaying in DataWindow 578
getting current 734
getting from id 735
getting id 737
hiding 1267
importing 784, 788, 793
in primary buffer 1157
inserting 836
modification status 705, 721, 766, 939, 1286
moving 1162
refreshing timestamp columns 1130
replacing text 1333
reporting number not displayed 580
retrieving data from 694, 697, 699, 702, 707, 709
retrieving from database 1146
scrolling 1183, 1186, 1191, 1194
selecting 739, 859, 1218
setting current 1313
setting height 1267
setting value of 1280
sorting 1377
under pointer 725

updating 1456
validating 756
rows, database
deleting 165, 166
fetching 169
inserting 170
updating 175
updating cursored row 178
RowsCopy function 1159
RowsDiscard function 1161
RowsMove function 1162
RPC 99
Run function 1164

S

Save As dialog box 1169, 1172
Save event 355
Save File response window 679
Save function 1166
SaveAsAscii function 1178
SaveDocument function 1180
scatter graphs
adding values to series 393
changing data point values 956
importing data 785, 790, 791, 795
inserting data from strings 796
obtaining data point values 651
scope operator 118
screen
changing position relative to 959
display 676
distance to workspace 1469, 1470
printing 1086
scripts
last statement 1234
stopping execution 1145
terminating 151
triggering events 1444
Scroll function 1182
ScrollHorizontal event 356, 931
scrolling
ListBox 1345
TreeView 1275
scrolling functions
Scroll 1182
ScrollNextPage 1183
ScrollNextRow 1186
ScrollPriorPage 1189
ScrollPriorRow 1191
ScrollToRow 836, 1194
Top 1426
ScrollNextPage function 1183
ScrollNextRow function 1186
ScrollPriorPage function 1189
ScrollPriorRow function 1191
ScrollToRow function 1194
ScrollVertical event 358, 931
searching
rich text 586, 605
rows 582
Second function 1197
secondary DataWindow control 1360, 1361, 1363
SecondsAfter function 1198
Seek function 1199
SeekType enumerated data type 1199
SELECT statement 173
SELECTBLOB statement 174
Selected event 359, 1295
SelectedColumn function 1201
SelectedIndex function 1202
SelectedItem function 1203
SelectedLength function 1204
SelectedLine function 1206
SelectedPage function 1208
SelectedStart function 1209
SelectedText function 1211
selection
clearing in list 1214
of rows 859
SelectionChanged event 360
SelectionChanging event 363
SelectItem function 1213
SelectObject function 1217
SelectRow function 1218
SelectText function
about 1220
copying to clipboard 466
SelectTextAll function 1225
SelectTextLine function 1226
SelectTextWord function 1227

Send function 1229
sender 906
SendMessage function 1229
series, graphs
  adding to 403
  adding values to 391
  clicked 965
  counting 1231
  data points 480, 507, 651, 667, 955, 1137
  deleting 519, 1133
  finding number of 610
  importing 785, 790, 795
  inserting 837
  inserting data 812
  obtaining name 1232
  reporting appearance of 740
  setting style 1317
SeriesCount function 1231
SeriesName function 1232
server application
  activating 385, 1217
  closing DDE channel 446
  connecting to 454, 456, 458, 459, 461
  DDE support 988
  pasting and linking 1027
  providing data 730
  sending data to 1310
  sending to DDE client 1256
  sending verb to 968
  stopping 1403
SetActionCode function 1234
SetAlignment function 1236
SetArgElement function 1237
SetAutomationPointer function 1241
SetAutomationTimeout function 1243
SetBorderStyle function 1245
SetChanges function 1246
SetColumn function 1249
SetConnect function 1252
SetData function 1254
SetDataDDE function 1256
SetDataPieExplode function 1258
SetDataStyle function 1260
SetDetailHeight function 1267
SetDropHighlight function 1269
SetDynamicParm function 1270
SetFilter function 1272
SetFirstVisible function 1275
SetFocus function 1276
SetFormat function 1277
SetFullState function 1278
SetItem function 1280
SetItemStatus function 1286
SetLevelPictures function 1289
SetLibraryList function 1291
SetMask function 1293
SetMicroHelp function 1295
SetNull function 1296
SetOverlayPicture function 1297
SetPicture function 1300
SetPointer function 1301
SetPosition function 1303
SetProfileString function 1306
SetRedraw function 1308
SetRemote function 1310
SetRow function 1313
SetRowFocusIndicator function 1315
SetSeriesStyle function 1317
SetSort function 1325
SetSQLPreview function 1328
SetSQLSelect function 1329
SetState function 1331
SetTabOrder function 1332
SetText function 1333
SetToolbar function 1338
SetTop function 1345
SetTraceFileName function 1346
SetTrans function 1348
SetTransObject function 1350
SetTransPool function 1354
setup printer 1087
SetValidate function 1356
SetValue function 1358
shade
  data points 660, 1261
  series 741, 1317
ShadowBox border style 627
shapes
  mouse pointer 1301
  printing 1079, 1082, 1084
shared objects
  SharedObjectDirectory function 1364

SharedObjectGet function 1365
SharedObjectRegister function 1367
SharedObjectUnregister function 1368
shared variables
about 36, 37
initialized 44
ShareData function 1360
ShareDataOff function 1363
SharedObjectDirectory function 1364
SharedObjectGet function 1365
SharedObjectRegister function 1367
SharedObjectUnregister function 1368
sharing data 481, 1360
sheets
arranging 408
getting active 619
getting first open 681
getting next open 723
obtaining parent 1023
opening 972, 990, 993
toolbars 759, 761, 1338, 1340
Show event 365
Show function 1369
ShowHeadFoot function 1370
ShowHelp function 1371
Sign function 1373
SignalError function 1374
Sin function 1376
sine 1376
SingleLineEdit functions
CanUndo 421
Clear 432
Copy 465
Cut 478
Move 959
Paste 1025
Position 1041
ReplaceText 1128
SelectedLength 1204
SelectedStart 1209
SelectedText 1211
SelectText 1220
Undo 1454
size
changing 1142
of screen 676
of string or blob 869
solid fill pattern 1265, 1322
Sort event 366
Sort function 1377
sort order
and GetCalc function 775
sharing data 1360
specifying criteria 1325
when inserting items into lists 819
SortAll function 1381
sounds (beep) 412
source database 1385
Space function 1383
spaces
deleting leading 868
deleting trailing 1154
inserting in a string 1383
removing from strings 1448
special ASCII characters in strings 9
Specify filter dialog box 1272
Specify Sort Columns dialog 1325
SQL Anywhere 950
SQL server 1411
SQL statements
about 155
and modification status 705
and SetTrans function 1348
and SetTransObject function 1350
and Update function 1456
changing during execution 1328, 1329
CLOSE Cursor 158
CLOSE Procedure 159
COMMIT 160
CONNECT 161, 1146
continuing 19
DataWindow source code from SELECT 1410
DECLARE Procedure 163
DISCONNECT 167
error handling 156
EXECUTE 168, 1270
FETCH 169
in pipeline execution 1386
INSERT 170
modifying WHERE clause of SELECT 941
OPEN 1270
OPEN Cursor 171

- painting 157
- previewing 751, 752
- ROLLBACK 172
- saving DataWindow SQL 1168
- SELECT 173
- SELECT and sharing data 1360
- SELECT, obtaining 524
- SELECTBLOB 174
- specifying retrieval arguments 1146
- UPDATE 175
- UPDATE Where Current of Cursor 178
- UPDATEBLOB 176

SQLCA 1350
SQLCode property 156
SQLDBCode property 156
SQLErrText property 156
SQLPreview event 369, 751, 766, 1328
Sqrt function 1384
square fill pattern 1265, 1322
square root 1384
stack faults
- and AcceptText function 383
- avoiding 1250, 1457

Start function
- about 1385
- canceling pipeline 420
- server application 458, 461

StartHotLink function 1393
StartServerDDE function 1395
state
- of listbox items 1397
- setting highlighted 1331

State function 1397
statements, PowerScript
- assignment 128
- CALL 131
- CHOOSE CASE 132
- CONTINUE 134
- CREATE 135
- DESTROY 139
- DO...LOOP 140
- EXIT 143
- FOR...NEXT 144
- GOTO 146
- HALT 147
- IF...THEN 148
- listed 127
- RETURN 151
- separating 21

StaticText control, inserting clipboard 435
status
- changing 1140, 1286
- of rows and columns 705, 766

stgShareMode enumerated data type 978, 981, 982
Stop function 1399
stop sign icon 930
StopHotLink function 1400
StopListening function 1402
StopServerDDE function 1403
storages, OLE
- file 1172
- releasing 439
- saving 1166

stored procedures
- closing 159
- declaring 157, 163
- executing 168

streams, OLE
- checking 926
- deleting 924
- renaming 928

string data type 26
String function 1404
string functions
- Asc 410
- Char 426
- Fill 577
- LeftTrim 868
- Len 869
- Lower 894
- Match 919
- Mid 933
- Pos 1039
- Replace 1126
- Right 1153
- RightTrim 1154
- Space 1383
- Trim 1448
- Upper 1462

StringParm property 994, 1001, 1003, 1010, 1012
strings
- char arrays 83

comparing 76
concatenating 78
continuing 19
converting 410, 413, 484, 503, 540, 893, 1106
converting to char 83
deleting leading spaces 868
detecting contents 853, 857, 861
determining width for printing 1094
extracting 426, 933
finding substrings 1039
getting dynamic 674
importing data from 793
lowercase 894
nested 27
reading a stream into 1103
retrieving from buffers 707
uppercase 1462
writing to stream 1471
structure objects
exporting as syntax 879
listing 877
recreating from syntax 881
structure of DataWindow 524
structures
about 86
assignment 93
autoinstantiated user objects 92
for return values 448
mailRecipient 911
passing as arguments 113
passing to external functions 69
passing values as 1015, 1017
style, border 627
substorages, OLE
checking 926
deleting 924
renaming 928
saving 1172
substrings
extracting 933
finding 1039
replacing 1126
subtraction operator
about 74
surrounded by spaces 22, 74
summary, moving objects to 1305
Super pronoun 17
symbol types, graphs
data points 663, 1264
series 1321
syntax
exporting object as 879
for creating objects 954
generating DataWindow source code 1410
recreating objects from 881
SyntaxFromSQL function 1410
system
base class 88
date 1425
events 196, 1047
events, defined 98
functions 118
object classes 88
object data types 32
object hierarchy 32
registry 1108, 1109, 1111, 1113, 1116
time 964
system and environment functions
Clipboard 435
CommandParm 452
DebugBreak 502
DoScript 538
FindClassDefinition 590
FindFunctionDefinition 592
FindTypeDefinition 612
GarbageCollect 614
GarbageCollectGetTimeLimit 615
GarbageCollectSetTimeLimit 616
GetApplication 621
GetEnvironment 676
Handle 775
PopulateError 1037
Post 1047
ProfileInt 1097
ProfileString 1099
Restart 1145
Run 1164
Send 1229
SetProfileString 1306
ShowHelp 1371
SignalError 1374
Yield 1474

SystemError event 372
SystemKey event 373
SYSTEMREAD modifier 47
SystemRoutine function 1413
SYSTEMWRITE modifier 47

T

tab character, specifying 9
Tab functions
  CloseTab 444
  MoveTab 961
  SelectTab 1219
  TabPostEvent 1414
  TabTriggerEvent 1415
tab order 1332
tab pages
  changing order 961
  CreatePage function 476
  opening user objects 996, 1000
  PageCreated function 1022
  selecting 1219
tables, database
  accessing multiple 1349
  changing update status 941
  names 1329
  transferring data between databases 1385
  updating multiple 948
Tabular presentation style 1410
Tag property
  and GetFocus function 684
  storing MicroHelp text 1295
Tan function 1416
tangent 1416
target database for pipeline 1385
temporary files 906
terminator for string 415
text
  deleting from edit controls 432
  finding in RichTextEdit 582, 605
  finding substrings 1039
  importing data from string 793
  line spacing when printing 1061
  metacharacters 919
  MicroHelp 1295
  obtaining current line 1417, 1418
  of listbox item 1203
  of message box 930
  on clipboard 435, 466, 478
  pasting over 1026
  printing 1060, 1092
  replacing 1128, 1333
  restoring 1454
  save rich text as ASCII 1180
  selecting 1204, 1211, 1220
  setting color of 1151
text file
  converting to Macintosh format 1099
  importing data from 788
  saving to 1168, 1170
Text function 1417
Text property 684
TextLine function 1418
This pronoun 16
tilde
  about 943
  in strings 27
  rules for 28
  specifying 9
time
  checking string 861
  converting to data type 1419
  CPU 470
  DateTime data type 487
  getting dynamic 671, 675
  minutes 937
  now 964
  relative 1118
  retrieving data from 697
  retrieving from buffers 709
  seconds 1197, 1198
time data type 28
Time function 1419
Timer event 375
Timer function 1422
timers, triggering event 1422
timestamps 1130
timing functions
  CPU 470
  Idle 782
  Timer 1422

timing object
  starting 1387
  stopping 1399
title of message box 930
ToAnsi function 1424
Today function 1425
ToolbarMoved event 377
toolbars 759, 761, 1338, 1340
top
  bringing object to 1369
  determining distance from 1034
  moving listbox item to 1345
  moving objects to 1305
Top function 1426
topics
  calling Help 1371
  ending server application 1403
  starting server application 1395
TotalColumns function 1427
TotalItems function 1428
TotalSelected function 1429
ToUnicode function 1430
Trace file functions, Open 970
TraceBegin function 1431
TraceClose function 1433
TraceDisableActivity function 1434
TraceEnableActivity function 1436
TraceEnd function 1438
TraceError function 1439
TraceFile objects
  Close function 441
  NextActivity function 962
  Reset function 1133
TraceOpen function 1440
TraceTree objects
  BuildModel function 417
  DestroyModel function 530
  EntryList function 551
  SetTraceFileName function 1346
TraceTreeGarbageCollect objects, GetChildrenList function 634
TraceTreeObject objects, GetChildrenList function 634
TraceTreeRoutine objects, GetChildrenList function 634
TraceUser function 1443
tracing functions
  TraceBegin 1431
  TraceClose 1433
  TraceDisableActivity 1434
  TraceEnableActivity 1436
  TraceEnd 1438
  TraceError 1439
  TraceOpen 1440
  TraceUser 1443
trailer
  locating 625
  moving objects to 1305
Transaction object functions
  DBHandle 501
  SyntaxFromSQL 1410
Transaction objects
  and Update function 1457
  creating 135
  getting values of 764
  resetting 1139
  setting values of 1348
  specifying 1350
  specifying before row retrieval 1146
transparent line style, graphs
  setting for data points 1263
  setting for series 1320
Transport objects
  Listen function 889
  StopListening function 1402
TreeView functions
  AddPicture 401
  CollapseItem 451
  DeleteItem 512
  DeletePicture 516
  DeletePictures 517
  DeleteStatePicture 522
  DeleteStatePictures 523
  EditLabel 549
  ExpandAll 560
  ExpandItem 561
  FindItem 599
  GetItem 692
  InsertItem 822, 823
  InsertItemFirst 825
  InsertItemLast 828
  InsertItemSort 831
  SelectItem 1216
  SetDropHighlight 1269

SetFirstVisible 1275
SetItem 1284
SetLevelPictures 1289
SetOverlayPicture 1297
Sort 1378
SortAll 1381
TrigEvent enumerated data type 1049
TriggerEvent function 1444
triggering
events 196
functions or events 102
TriggerPBEvent function 1446
Trim function 1448
Truncate function 1449
TypeOf function 1450

U

Uncheck function 1452
underline border style 627
Undo
providing capability 1163
testing 421
Undo function 1454
Unicode 1424, 1430
Uniform Data Transfer 653, 1254
units
converting from pixels 1032
converting to pixels 1455
distance from edge 1033
UnitsToPixels function 1455
unread messages 900
unsigned integer data type 28
unsigned long data type 28
update flags 1140
Update function 1456
UPDATE statement 175
update status
after row copy 1160
and Update function 705
changing 941, 1286
resetting flags 1140
UPDATE Where Current of Cursor statement 178
UPDATEBLOB statement 176
UpdateEnd event 378
UpdateStart event 379
Upper function 1462
UpperBound function 1463
uppercase 1462
user events
defined 98
pbm_dwngraphcreate 1318
user ID 904
user name 911
User objects
about 88
autoinstantiated 92
closing 446
closing tab page 444
creating 135
creating dynamically 136
exporting as syntax 879
listing 877
opening 996, 997, 998, 1005, 1006, 1008, 1009
pipeline 1385
recreating from syntax 881
tab pages 996, 1000
used like structures 92
user-defined events 196, 199

V

validation rules
and AcceptText function 383
and SetItem function 1281
checking on update 1457
obtaining 770
setting 1356
value, passing arguments by 112
values
adding to lists 394
checking for NULL 856
data points 667
deleting from list 510
detecting numeric 857
edit control 756
inserting into lists 818
obtaining column 771
setting item 1358
setting text in edit control 1333

variables
- access levels 45
- assigning literals 24, 25, 26, 28
- assigning values 43
- checking for NULL 856
- data type 42
- declaring 36
- declaring values 42
- default values 43
- determining data type of 429
- extracting data from a blob 415
- host 155
- in Modify function 943
- indicator 155
- initializing with expression 44
- inserting data into a blob 414
- names 42
- OLEObject 459
- referencing in SQL 155
- search order 37
- setting to NULL 11, 1296
- validating 863
- where to declare 36

variable-size arrays, memory allocation 55, 1463

VBX user object functions
- AddItem 394
- DeleteItem 510
- EventParmDouble 553
- EventParmString 554
- InsertItem 819

vertical fill pattern 1265, 1322

video monitor 676

ViewChange event 380

Visible property
- and SetRedraw function 1308
- displaying popup menus 1035
- setting 1369

visual user objects 88

W

warm link 555, 731, 987, 1311

week, day of 490, 491

WHERE clause 941, 944, 949, 950

white space 22

width
- data point's line 1262
- series line 1319
- setting 1142
- string 1094
- workspace 1468

Window ActiveX controls
- GetArgElement function 622
- GetLastReturn function 712
- InvokePBFunction function 845
- ResetArgElements function 1135
- SetArgElement function 1237
- TriggerPBEvent function 1446

Window functions
- ArrangeSheets 408
- ChangeMenu 425
- ClassName 429
- CloseUserObject 446
- Draw 546
- GetActiveSheet 619
- GetFirstSheet 681
- GetNextSheet 723
- Hide 778
- Move 959
- Open 970
- OpenSheet 990
- OpenSheetWith Parm 993
- OpenTab 996
- OpenUserObject 1005
- OpenWith Parm 1014
- ParentWindow 1023
- PointerX 1033
- PointerY 1034
- PostEvent 1049
- print 1058
- Resize 1142
- SetFocus 1276
- SetMicroHelp 1295
- SetPosition 1303
- SetRedraw 1308
- Show 1369
- TriggerEvent 1444
- TypeOf 1450
- WorkSpaceHeight 1466
- WorkSpaceWidth 1468
- WorkSpaceX 1469

WorkSpaceY 1470
Window objects
closing user objects 446
exporting as syntax 879
listing 877
recreating from syntax 881
Window painter
about 1005, 1007
Picture control in 1316
windows
adding user objects 996, 1005, 1009
arranging 408, 990
changing menus 425
closing 438
creating dynamically 471
custom frames 1469, 1470
data type of 970
DDE conversation handle 1395
getting active 619
obtaining handle 775
obtaining workspace height 1466
obtaining workspace width 1468
opening 970, 1014
posting messages 1047
setting position of 1303
WK1/WKS file 1168
WordParm field
and TriggerEvent function 1444
posting events 1049
workspace
distance to screen 1469, 1470
obtaining height of 1466
obtaining width 1468
WorkSpaceHeight function 1466
WorkSpaceWidth function 1468
WorkSpaceX function 1469
WorkSpaceY function 1470
Write function 1471
Writes 1471

X

x value
data point 651, 667, 956
importing data 785, 790, 791, 795
inserting from strings 796
xValue enumerated data type 651, 667

Y

y value
data point 651, 667, 956
importing data 785, 790, 791, 795
inserting from strings 796
Year function 1473
year, about 486
Yield function 1474
yValue enumerated data type 651, 667

Z

zero, determining 1373