Contents

About This Book ........................................................................................................................... xi

PART 1 PROGRAMMERS GUIDE

CHAPTER 1 Introduction to PBNI ................................................................. 3
  About PBNI ....................................................................................................................... 3
  Understanding PowerBuilder extensions ................................................................. 4
  Embedding the PBVM in a C++ application .............................................................. 6
  The elements of PBNI ................................................................................................. 6
  The PBNI SDK ................................................................................................................ 8
  Comparing PBNI and JNI ........................................................................................... 10

CHAPTER 2 Building PowerBuilder Extensions ............................................... 11
  Nonvisual extension example ..................................................................................... 11
  Building the pbadd PowerBuilder extension ............................................................ 12
  Using the extension in PowerBuilder ........................................................................ 15
  Creating a PowerBuilder extension ......................................................................... 17
  Step 1: Decide on a feature to implement ............................................................... 18
  Step 2: Define the classes and functions in the extension .................................... 18
  Step 3: Declare native classes and global functions ............................................. 21
  Step 4: Implement native classes and global functions ......................................... 22
  Step 5: Export methods to create class instances ............................................... 24
  Step 6: Build a PBX ..................................................................................................... 26
  Adding an extension to a PowerBuilder target ...................................................... 26
  Using the extension .................................................................................................... 27
  Creating and using a visual extension ..................................................................... 28
  Step 1: Decide on a feature to implement ............................................................... 28
  Step 2: Define the classes and functions in the extension .................................... 29
  Step 3: Declare visual classes and global functions ............................................. 29
  Step 4: Implement native classes ........................................................................... 29
  Step 5: Export methods to create class instances ............................................... 29
  Step 6: Build and use a PBX ...................................................................................... 30
  Step 7: Use the visual extension in an application ............................................... 30
Contents

Creating visual class instances ...................................................... 31
Event processing in visual extensions ............................................. 34
Using an event name with return type and arguments ................. 34
Using an event name with a PowerBuilder event ID .......... 37
Processing events sent to the parent of the window ............. 38
Calling PowerScript from an extension ................................. 39
Example: Calling PowerBuilder functions......................... 42
Exception handling and debugging ............................................. 43

CHAPTER 3 Creating Marshaler Extensions ................................. 45
About marshaler extensions....................................................... 45
Developing the PowerBuilder extension ................................. 46
Step 1: Describe the extension ............................................. 47
Step 2: Implement the creator class ...................................... 48
Step 3: Implement the marshaler class .................................. 51
Generating proxies for Java classes ......................................... 53
Calling the Java class from PowerBuilder .............................. 54

CHAPTER 4 Exchanging Data with PowerBuilder ......................... 55
About exchanging data with PowerBuilder ............................ 55
Passing values between extensions and the PBVM .................. 55
PBCallInfo structure .............................................................. 56
IPB_Arguments interface ......................................................... 56
IPB_Value interface ............................................................... 57
Using the IPB_Session interface ............................................. 59
Saving data from IPB_Value to a local variable ................. 61
Using variables throughout a session .................................. 63
Handling enumerated types ..................................................... 64

CHAPTER 5 Calling PowerBuilder from C++ ............................... 65
About calling PowerScript from C++ applications ................. 65
Calling PowerBuilder objects from C++ .............................. 66
Creating a PowerBuilder object to be called from C++ ....... 66
Getting the signature of a function ....................................... 67
Creating the C++ application .............................................. 68
Running the C++ application ............................................. 72
Accessing result sets ............................................................. 72
Processing PowerBuilder messages in C++ ...................... 73
Examples ........................................................................ 73
More PBNI possibilities ....................................................... 78

PART 2 REFERENCE
## CHAPTER 6 PBNI Types and Return Values ................................................... 83

- PowerBuilder to PNI datatype mappings ............................................. 83
- Types for access to PowerBuilder data ............................................. 84
- PNI enumerated types ..................................................................... 84
- Error return values ......................................................................... 86

## CHAPTER 7 PNI Interfaces, Structures, and Methods ................................. 87

- Header file contents ........................................................................ 88
- Class and interface summary .......................................................... 88

### IPB_Actions interface .................................................................... 90
- GetAt .......................................................................................... 90
- GetCount ................................................................................... 91

### IPB_ResultSetAccessor interface .................................................. 92
- AddRef ....................................................................................... 92
- GetColumnCount ........................................................................ 92
- GetColumnMetaData ..................................................................... 93
- GetItemData ................................................................................ 94
- GetRowCount .............................................................................. 94
- Release ....................................................................................... 95

### IPB_RSItemData interface ............................................................. 95
- SetData ........................................................................................ 95
- SetNull ........................................................................................ 96

### IPB_Session interface ..................................................................... 96
- AcquireArrayItemValue .............................................................. 102
- AcquireValue ............................................................................. 103
- Add<type>Argument ..................................................................... 104
- AddGlobalRef ............................................................................ 106
- AddLocalRef .............................................................................. 107
- ClearException .......................................................................... 107
- CreateResultSet ......................................................................... 108
- FindClass ................................................................................... 111
- FindClassByClassID ................................................................. 111
- FindGroup .................................................................................. 112
- FindMatchingFunction ............................................................... 112
- FreeCallInfo ............................................................................ 114
- Get<type>ArrayItem ................................................................... 114
- Get<type>Field .......................................................................... 116
- Get<type>GlobalVar ................................................................... 117
- Get<type>SharedVar .................................................................. 118
- GetArrayInfo ............................................................................. 120
- GetArrayItemType ....................................................................... 121
- GetArrayLength ......................................................................... 122
- GetBlob ..................................................................................... 123
- GetBlobLength .......................................................................... 124
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetClass</td>
<td>125</td>
</tr>
<tr>
<td>GetClassName</td>
<td>126</td>
</tr>
<tr>
<td>GetCurrGroup</td>
<td>126</td>
</tr>
<tr>
<td>GetDateString</td>
<td>127</td>
</tr>
<tr>
<td>GetDateTimeString</td>
<td>127</td>
</tr>
<tr>
<td>GetDecimalString</td>
<td>128</td>
</tr>
<tr>
<td>GetEnumItemName</td>
<td>128</td>
</tr>
<tr>
<td>GetEnumItemValue</td>
<td>129</td>
</tr>
<tr>
<td>GetException</td>
<td>130</td>
</tr>
<tr>
<td>GetFieldID</td>
<td>130</td>
</tr>
<tr>
<td>GetFieldName</td>
<td>131</td>
</tr>
<tr>
<td>GetFieldType</td>
<td>132</td>
</tr>
<tr>
<td>GetGlobalVarID</td>
<td>132</td>
</tr>
<tr>
<td>GetGlobalVarType</td>
<td>133</td>
</tr>
<tr>
<td>GetMarshaler</td>
<td>134</td>
</tr>
<tr>
<td>GetMethodID</td>
<td>135</td>
</tr>
<tr>
<td>GetMethodIDByEventID</td>
<td>136</td>
</tr>
<tr>
<td>GetNativeInterface</td>
<td>137</td>
</tr>
<tr>
<td>GetNumOfFields</td>
<td>138</td>
</tr>
<tr>
<td>GetPBAnyArrayItem</td>
<td>138</td>
</tr>
<tr>
<td>GetPBAnyField</td>
<td>139</td>
</tr>
<tr>
<td>GetPBAnyGlobalVar</td>
<td>141</td>
</tr>
<tr>
<td>GetPBAnySharedVar</td>
<td>141</td>
</tr>
<tr>
<td>GetProp</td>
<td>142</td>
</tr>
<tr>
<td>GetResultSetAccessor</td>
<td>142</td>
</tr>
<tr>
<td>GetSharedVarID</td>
<td>143</td>
</tr>
<tr>
<td>GetSharedVarType</td>
<td>144</td>
</tr>
<tr>
<td>GetString</td>
<td>145</td>
</tr>
<tr>
<td>GetStringLength</td>
<td>146</td>
</tr>
<tr>
<td>GetSuperClass</td>
<td>146</td>
</tr>
<tr>
<td>GetSystemClass</td>
<td>147</td>
</tr>
<tr>
<td>GetSystemGroup</td>
<td>147</td>
</tr>
<tr>
<td>GetTimeString</td>
<td>148</td>
</tr>
<tr>
<td>HasExceptionThrown</td>
<td>148</td>
</tr>
<tr>
<td>HasPBVisualObject</td>
<td>149</td>
</tr>
<tr>
<td>InitCallInfo</td>
<td>150</td>
</tr>
<tr>
<td>InvokeClassFunction</td>
<td>151</td>
</tr>
<tr>
<td>InvokeObjectFunction</td>
<td>152</td>
</tr>
<tr>
<td>IsArrayItemNull</td>
<td>153</td>
</tr>
<tr>
<td>IsAutoInstantiate</td>
<td>153</td>
</tr>
<tr>
<td>IsFieldArray</td>
<td>153</td>
</tr>
<tr>
<td>IsFieldNull</td>
<td>154</td>
</tr>
<tr>
<td>IsFieldObject</td>
<td>155</td>
</tr>
<tr>
<td>IsGlobalVarArray</td>
<td>155</td>
</tr>
</tbody>
</table>
Contents

SetFieldToNull ............................................................... 190
SetGlobalVarToNull ...................................................... 190
SetMarshaler ................................................................. 191
SetProp ........................................................................... 192
SetSharedVarToNull ...................................................... 194
SetString ......................................................................... 194
SetTime ............................................................................ 196
SetValue ............................................................................ 197
SplitDate ........................................................................... 197
SplitDateTime ................................................................. 198
SplitTime .......................................................................... 198
ThrowException .............................................................. 199
TriggerEvent ................................................................. 200
UpdateField ................................................................. 201
IPB_Value interface ........................................................... 202
  Get<type> ........................................................................ 203
  GetClass ......................................................................... 204
  GetType .......................................................................... 204
  isArray .......................................................................... 205
  IsByRef .......................................................................... 205
  IsEnum ........................................................................... 206
  IsNull ............................................................................. 206
  IsObject ........................................................................... 207
  Set<type> ........................................................................ 208
  SetToNull ....................................................................... 210
IPB_VM interface ............................................................... 211
  CreateSession ............................................................... 211
  RunApplication ............................................................. 213
IPBX_Marshaler interface ..................................................... 215
  Destroy .......................................................................... 215
  GetModuleHandle ........................................................... 216
  InvokeRemoteMethod ...................................................... 217
IPBX_NonVisualObject interface ......................................... 218
IPBX_UserObject interface .................................................. 219
  Destroy .......................................................................... 219
  Invoke .......................................................................... 220
IPBX_VisualObject interface ............................................... 221
  CreateControl ............................................................... 221
  GetEventID ..................................................................... 223
  GetWindowClassName ...................................................... 225
PBArrayInfo structure ....................................................... 226
PBCallInfo structure ........................................................ 226
PB_DateData structure ...................................................... 227
PB_DateTimeData structure ................................................. 227
PB_TimeData structure ................................................................. 227
PBX_DrawItemStruct structure .................................................... 228
PArrayAccessor template class .................................................. 229
GetAt ..................................................................................... 229
IsNull ..................................................................................... 229
SetAt ...................................................................................... 230
SetToNull............................................................................... 231
PBBoundedArrayCreator template class .................................... 232
GetArray ................................................................................ 232
SetAt ...................................................................................... 233
PBBoundedObjectArrayCreator class .......................................... 235
GetArray ................................................................................ 235
SetAt ...................................................................................... 235
PObjectArrayAccessor class ...................................................... 236
GetAt ..................................................................................... 236
SetAt ...................................................................................... 237
PBUnboundedArrayCreator template class ................................ 238
GetArray ................................................................................ 238
SetAt ...................................................................................... 239
PBUnboundedObjectArrayCreator class ...................................... 240
GetArray ................................................................................ 240
SetAt ...................................................................................... 240
Exported methods ........................................................................ 241
PBX_CreateNonVisualObject................................................ 241
PBX_CreateVisualObject ...................................................... 243
PBX_DrawVisualObject......................................................... 244
PBX_GetDescription ............................................................. 246
PBX_InvokeGlobalFunction .................................................. 249
PBX_Notify ............................................................................ 251
Method exported by PowerBuilder VM ....................................... 252
PB_GetVM ............................................................................ 252

CHAPTER 8 PNI Tool Reference ......................................................... 255
pbsig125 ....................................................................................... 255
pbx2pbd125 ................................................................................. 259

PART 3 APPENDIX

APPENDIX A Using the Visual Studio Wizards ................................. 263
Where the wizards are installed .................................................. 263
Generating a PNI project ........................................................... 265
Setting project options ............................................................. 266
Contents

Building and using the PBX.......................................................... 266

Index ........................................................................................................................................... 267
### About This Book

**Audience**
This book is for C++ programmers who will use the PowerBuilder® Native Interface (PBNI) to build PowerBuilder extensions. The book assumes that you are familiar with the C++ language and a C++ development tool.

**Related documents**
This book contains information about building PowerBuilder extensions. The *PowerBuilder Extension Reference* contains information about using extensions that are provided with PowerBuilder.

**Other sources of information**
Use the Sybase® Getting Started CD and the Sybase Product Documentation Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.

- The Sybase Product Documentation Web site is accessible using a standard Web browser. In addition to product documentation, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.


**Conventions**
The formatting conventions used in this manual are:

<table>
<thead>
<tr>
<th>Formatting example</th>
<th>To indicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve and Update</td>
<td>When used in descriptive text, this font indicates:</td>
</tr>
<tr>
<td></td>
<td>• Command, function, and method names</td>
</tr>
<tr>
<td></td>
<td>• Keywords such as true, false, and null</td>
</tr>
<tr>
<td></td>
<td>• Datatypes such as integer and char</td>
</tr>
<tr>
<td></td>
<td>• Database column names such as emp_id and t_name</td>
</tr>
<tr>
<td></td>
<td>• User-defined objects such as dw_emp or w_main</td>
</tr>
</tbody>
</table>
### Formatting example

<table>
<thead>
<tr>
<th>variable or file name</th>
<th>To indicate</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Variable</em> or <em>file name</em></td>
<td>When used in descriptive text and syntax descriptions, oblique font indicates:</td>
</tr>
<tr>
<td></td>
<td>• Variables, such as <em>myCounter</em></td>
</tr>
<tr>
<td></td>
<td>• Parts of input text that must be substituted, such as <em>pblname.pscl</em></td>
</tr>
<tr>
<td></td>
<td>• File and path names</td>
</tr>
</tbody>
</table>

| File>Save | Menu names and menu items are displayed in plain text. The greater than symbol (>) shows you how to navigate menu selections. For example, File>Save indicates “select Save from the File menu.” |

<table>
<thead>
<tr>
<th>dw_1.Update()</th>
<th>Monospace font indicates:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Information that you enter in a dialog box or on a command line</td>
</tr>
<tr>
<td></td>
<td>• Sample script fragments</td>
</tr>
<tr>
<td></td>
<td>• Sample output fragments</td>
</tr>
</tbody>
</table>

### If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the documentation or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.
This part provides an introduction to the PowerBuilder Native Interface and a guide to creating PowerBuilder extensions and interacting with PowerBuilder.
CHAPTER 1

Introduction to Pdni

About this chapter
This chapter provides a brief introduction to the PowerBuilder Native Interface.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About Pdni</td>
<td>3</td>
</tr>
<tr>
<td>The elements of Pdni</td>
<td>6</td>
</tr>
<tr>
<td>The Pdni SDK</td>
<td>8</td>
</tr>
<tr>
<td>Comparing Pdni and JNI</td>
<td>10</td>
</tr>
</tbody>
</table>

About Pdni

Pdni is a standard programming interface that enables developers to extend the functionality of PowerBuilder. Using Pdni, you can create extensions to PowerBuilder—nonvisual, visual, and marshaler extensions—and embed the PowerBuilder virtual machine (PBVM) into C++ applications. Through the Java Native Interface (JNI) and Pdni, Java applications can also communicate with the PBVM.

Use with .NET targets
You can use the built-in Web services client extension (pbwsclient125.pbx) in applications that you plan to deploy to .NET as a PowerBuilder .NET Windows Forms application. You cannot use any other Pdni extensions in a .NET target.

Code samples
This documentation contains two complete but very simple examples that illustrate some basic principles of using the PowerBuilder Native Interface (Pdni): “Nonvisual extension example” on page 11 and “Creating a PowerBuilder object to be called from C++” on page 66. For more real-world examples, see the Pdni section of the PowerBuilder CodeXchange Web site at http://powerbuilder.codelxchange.sybase.com/.
The following diagram illustrates the two-way communication, with both PowerBuilder extensions and external applications, that PBNI provides for the PBVM. As the diagram shows, a PowerBuilder extension communicates with the PBVM through the IPB_Session interface, and the PBVM communicates with the extension through an interface derived from IPBX_UserObject.

C++ and Java extensions communicate with the PBVM through the IPB_VM and IPB_Session interfaces.

Figure 1-1: Interaction between the PBVM and external applications and extensions

Understanding PowerBuilder extensions

A PowerBuilder extension is just what its name suggests: an extension to PowerBuilder functionality provided by you, by a third party, or by Sybase. All PowerBuilder extensions communicate with the PBVM through an interface called IPB_Session. This interface and other PBNI objects and interfaces are described in “The elements of PBNI” on page 6.

PowerBuilder provides its own extensions, including a PBDOM XML parser and classes that support SOAP clients for Web services. In future releases, Sybase might develop more new features as PBNI extensions instead of embedding them in the PowerBuilder VM (PBVM), so that the size of the PBVM can be minimized. Extensions are also available from third party contributors; for the latest samples and utilities, see the PBNI section of the PowerBuilder CodeXchange Web site at http://powerbuilder.codeXchange.sybase.com/.

Nonvisual extensions

The most frequently used type of PowerBuilder extension is a nonvisual extension. Nonvisual extensions provide a way to call C and C++ functions from PowerBuilder with more flexibility than the previous solution of declaring a function in a script. They also allow you to use object-oriented techniques when working with external objects.
A nonvisual extension is a DLL, written in C++, that exposes one or more native classes and/or global functions. Classes are used in a PowerBuilder application as though they were class user objects created in PowerBuilder—a native class is simply a PowerScript class that is implemented in C++. Global functions in an extension are used like global functions declared in the Function painter.

Nonvisual extensions allow you to use datatypes in C++ that map to standard PowerBuilder datatypes. PBNI provides predefined datatypes that map to PowerBuilder datatypes, so that you can use PowerBuilder datatypes when you invoke the methods of the native class, and the native class can use predefined types to call back into PowerBuilder. For more information about predefined types, see Chapter 6, “PBNI Types and Return Values.”

You can use native classes to call back into the PBVM from the C++ code and trigger PowerBuilder events and invoke functions. You can also call external functions that require callback functions. For example, if your PowerBuilder application uses an extension that is a SAX XML parser, the SAX parser can send information back to the PowerBuilder application about the items it has encountered in the XML document that it is parsing. In response, the PowerBuilder application can send back instructions on how to handle those items.

Possible uses for a nonvisual extension include:

- A wrapper for a Component Object Model (COM) component that references a user-defined COM interface that cannot be mapped to a PowerBuilder datatype
- A PowerBuilder interface for database backups and administration using the SQL Anywhere™ dbtools (which require callback functions)
- Wrappers for any open source C++ libraries that provide standard utilities

PowerBuilder extensions run faster than standard PowerBuilder user objects because they are compiled in native machine code instead of PowerBuilder pseudocode (Pcode). PBNI complies with the C++ specification, so well-programmed code is portable at the source code level.

Visual extensions can be used as if they were PowerBuilder visual user objects—you can place them in windows or on other visual controls. Visual extensions allow you to create a subclass of the Windows procedure (winproe) of a visual component so that you can use the latest “look and feel” for your applications.
Marshaler extensions
Marshaler extensions act as bridges between PowerBuilder and other components, such as Enterprise JavaBeans (EJB) components, Java classes, Web services, and CORBA components. PowerBuilder provides a marshaler extension for creating clients for EJB components running in any J2EE-compliant application server. Other techniques for calling EJBs from PowerBuilder do not provide a standard way to marshal PowerBuilder requests to other components and unmarshal the result back to PowerBuilder.

Embedding the PBVM in a C++ application
Many PowerBuilder users have developed sophisticated custom class user objects that handle intensive database operations or other functionality. Such objects can already be used in external applications. However, limitations on the use of some datatypes and of overloaded functions, as well as other coding restrictions, diminishes the value of this technique.

To have direct access to a custom class user object running in the PBVM, and to take advantage of Pdni functions for data access and exchange, you can load the PBVM in the C++ application, create a session, and invoke the custom class user object’s functions from the external application.

Communication between the PBVM and a C++ application is based primarily on two interfaces: IPB_VM and IPB_Session.

Interacting with Java
To call Java classes from PowerBuilder, you can build a marshaler extension that invokes Java methods through JNI, as described in Chapter 3, “Creating Marshaler Extensions.” You can also use JNI to allow Java to call into PowerBuilder through C or C++. For an example, see the PowerBuilder CodeXchange Web site at http://powerbuilder.codexchange.sybase.com.

The elements of Pdni
To enable the features described in the previous section, Pdni provides interfaces, structures, global functions, and helper classes. These elements are described in more detail in the reference section of this guide. See Chapter 7, “Pdni Interfaces, Structures, and Methods.” This section provides an overview.

Interfaces
The IPB_VM interface is used to load PowerBuilder applications in third-party applications and interoperate with the PowerBuilder virtual machine (PBVM).
IPB_Session is an abstract interface that defines methods for performing various actions such as accessing PowerScript data, creating PowerBuilder objects, and calling PowerScript functions.

The IPB_Value and IPB_Arguments interfaces enable you to pass values between the PowerBuilder VM and PowerBuilder extension modules.

The IPB_Value interface represents a PowerBuilder value, which could be one of the PowerBuilder standard datatypes such as integer, long, string, and so forth. It provides information about each variable, including its type, null flag, access privileges, array or simple type, and reference type.

The IPB_Arguments interface represents the arguments passed to a PowerScript function and is used to access the data.

The IPB_ResultSetAccessor and IPB_RSItemData interfaces enable you to access data in a DataWindow or DataStore.

All PowerBuilder native classes inherit from the IPBX_NonVisualObject interface or the IPBX_VisualObject interface, which in turn inherit from the IPBX_UserObject interface. You must implement the Invoke method in the inherited class to enable PowerBuilder to invoke methods in the native class.

Marshaler extensions contain a class that inherits from the IPBX_Marshaler interface. You must implement the InvokeRemoteMethod method in the inherited class to enable PowerBuilder to invoke methods on remote objects represented by a proxy.

Structures

The PBCallInfo structure holds arguments and return type information for function calls between PBNI and PowerBuilder. To access the information in PBCallInfo, use the IPB_Arguments interface.

The PBArrayInfo structure stores information about arrays.

The PB_DateData, PB_TimeData, and PB_DateTimeData structures are used to pass DataWindow and DataStore data.

Global functions

Every PowerBuilder extension object must export global functions that enable the PowerBuilder VM to create instances of the object and use its methods. The PBX_GetDescription function describes the classes and functions in the extension. The PBX_CreateNonVisualObject function enables the PBVM to create instances of the nonvisual classes in an extension, and the PBX_CreateVisualObject function does the same for visual classes.

Helper classes

Several helper classes, such as PBOBJECTCreator, PBArraayAccessor, and PBEEvenTrigger, make it easier to program with PBNI.
The PNI SDK

Interaction between an extension and the PBVM

The following diagram summarizes how an extension interacts with the PBVM.

Figure 1-2: Interaction between an extension and the PowerBuilder VM

The PNI SDK

When you install PowerBuilder, the Software Development Kit (SDK) for PNI is installed in the PowerBuilder 12.50\SDK\PNI directory. The SDK tools, pbsig125 and pbx2pbd125, are also installed in the Shared\PowerBuilder directory so that they are available in your path.

The SDK contains the components shown in the following table.

Table 1-1: Contents of the PNI SDK

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbx2pbd125.exe</td>
<td>A tool that generates a PBD file from a PowerBuilder extension file. The extension file is a DLL file that must export a set of PNI functions. The DLL is usually called a PBX and can be given the suffix .pbx.</td>
</tr>
<tr>
<td>pbsig125.exe</td>
<td>A tool that generates a set of strings representing the return type and arguments of each function in a PBL. Use these strings to call PowerBuilder functions from external modules.</td>
</tr>
<tr>
<td>include\pbni.h</td>
<td>A header file that defines the structures and interfaces used to build PowerBuilder extensions.</td>
</tr>
<tr>
<td>include\parray.h</td>
<td>A header file that contains helper classes that make it easier to create arrays and access data in them.</td>
</tr>
</tbody>
</table>
### Component Description

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include\pbfield.h</code></td>
<td>A header file that contains helper classes that make it easier to access data in fields.</td>
</tr>
<tr>
<td><code>include\pbtraits.h</code></td>
<td>A header file used by <code>pbarray.h</code> and <code>pbfield.h</code> that provides specializations for the <code>pbvaluetype</code> enumerated types.</td>
</tr>
<tr>
<td><code>include\pbext.h</code></td>
<td>A header file that defines the functions that PowerBuilder extension functions must export.</td>
</tr>
<tr>
<td><code>include\pbevtid.h</code></td>
<td>A header file that maps the PowerBuilder event IDs to event names for use in visual extensions.</td>
</tr>
<tr>
<td><code>include\pbnimd.h</code></td>
<td>A header file that defines machine-dependent datatypes used in <code>pbni.h</code>.</td>
</tr>
<tr>
<td><code>include\pbrsa.h</code></td>
<td>A header file that defines interfaces and structures used to access <code>DataWindow</code> and <code>DataStore</code> data.</td>
</tr>
<tr>
<td><code>src\pbarray.cpp</code></td>
<td>A source file that must be added to your project if you want to use the following helper classes defined in <code>pbarray.h</code>:</td>
</tr>
<tr>
<td></td>
<td>- PBArrayAccessor</td>
</tr>
<tr>
<td></td>
<td>- PBOBJECTARRAYACCESSOR</td>
</tr>
<tr>
<td></td>
<td>- PBoundedArrayCreator</td>
</tr>
<tr>
<td></td>
<td>- PBBoundedObjectArrayCreator</td>
</tr>
<tr>
<td></td>
<td>- PBUndboundedArrayCreator</td>
</tr>
<tr>
<td></td>
<td>- PBUndboundedObjectArrayCreator</td>
</tr>
<tr>
<td><code>src\phfuninv.cpp</code></td>
<td>A source file that must be added to your project if you want to use the following helper classes defined in <code>pbni.h</code>:</td>
</tr>
<tr>
<td></td>
<td>- PBGlobalFunctionInvoker</td>
</tr>
<tr>
<td></td>
<td>- PBObjectFunctionInvoker</td>
</tr>
<tr>
<td></td>
<td>- PBEventTrigger</td>
</tr>
<tr>
<td><code>src\phobject.cpp</code></td>
<td>A source file that must be added to your project if you want to use the following helper class defined in <code>pbni.h</code>:</td>
</tr>
<tr>
<td></td>
<td>- PBObjectCreator</td>
</tr>
<tr>
<td><code>wizards\VCProjects 8.0</code></td>
<td>A Microsoft Visual Studio 2005 wizard that makes it easier for you to create <code>PBNI</code> projects.</td>
</tr>
<tr>
<td><code>wizards\VCProjects 7.1</code></td>
<td>A Microsoft Visual Studio .NET 2003 wizard that makes it easier for you to create <code>PBNI</code> projects.</td>
</tr>
<tr>
<td><code>wizards\VCProjects 7.0</code></td>
<td>A Microsoft Visual Studio .NET 2002 wizard that makes it easier for you to create <code>PBNI</code> projects.</td>
</tr>
<tr>
<td><code>wizards\VCWizard</code></td>
<td>Files required by the Visual Studio wizards.</td>
</tr>
<tr>
<td><code>pbni125.hlp</code>, <code>pbni125.cnt</code></td>
<td>Help files for <code>PBNI</code>.</td>
</tr>
</tbody>
</table>
Comparing PBNI and JNI

If you have used the Java Native Interface (JNI), which allows Java applications and C and C++ modules to interoperate, you might find it helpful to be aware of the similarities in the two interfaces and the differences between them.

The IPB_VM interface in PBNI is analogous to the JavaVM type, and the IPB_Session interface in PBNI is analogous to JNIEnv. For JNI, you use the javap command to obtain a string that encodes the signature of each method in a native class. For PBNI, the pbsig125 tool performs the same function.

The major difference between the two interfaces is in how a native function or class is declared.

In JNI, you must use the native keyword to declare that a function is native, but you cannot simply declare a class as native. You must define your classes in Java source code, use the javah tool to generate a C header file that defines a C prototype for each native method, then implement the individual C or C++ functions, using #include to include the generated header file.

PBNI provides an object-oriented approach—you declare a class as native in the C++ code by inheriting from the IPBX_NonVisualObject or IPBX_VisualObject struct.
CHAPTER 2

Building PowerBuilder Extensions

About this chapter
This chapter describes how to build a PowerBuilder extension. It begins with a sample application that uses a simple nonvisual extension.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonvisual extension example</td>
<td>11</td>
</tr>
<tr>
<td>Creating a PowerBuilder extension</td>
<td>17</td>
</tr>
<tr>
<td>Adding an extension to a PowerBuilder target</td>
<td>26</td>
</tr>
<tr>
<td>Using the extension</td>
<td>27</td>
</tr>
<tr>
<td>Creating and using a visual extension</td>
<td>28</td>
</tr>
<tr>
<td>Creating visual class instances</td>
<td>31</td>
</tr>
<tr>
<td>Event processing in visual extensions</td>
<td>34</td>
</tr>
<tr>
<td>Calling PowerScript from an extension</td>
<td>39</td>
</tr>
<tr>
<td>Exception handling and debugging</td>
<td>43</td>
</tr>
</tbody>
</table>

Nonvisual extension example

To illustrate the principles involved in building and using an extension, this chapter starts with a sample application that uses a PowerBuilder extension to perform a simple arithmetic operation. Ordinarily, this is not a task that needs PBNI, but it is used here to make the basic process clear. The rest of this chapter describes building extensions in more detail.

PBX file suffix
PowerBuilder extensions are DLL files but typically use the file extension .pbx instead of .dll. Your extension is compiled into a PBX file by default if you use the wizard described in the Appendix, “Using the Visual Studio Wizards.”

The following sample application has two main steps:

- Building the pbadd PowerBuilder extension
- Using the extension in PowerBuilder

### Building the pbadd PowerBuilder extension

In this example, the C++ code is in three files:

- The class declaration is in a header file, `pbadd.h`
- The standard functions that every PowerBuilder extension must expose are in `main.cpp`
- The implementation of the class is in `pbadd.cpp`.

#### To implement the pbadd extension:

1. Create the `pbadd.h` header file.

The `pbadd.h` header file declares the pbadd class. The file includes `pbext.h`, which must be included in all PowerBuilder extensions because it declares the ancestor classes for native classes and the standard functions that the extension must expose. Here is the code for `pbadd.h`:

```cpp
#include "pbext.h"

class pbadd: public IPBX_NonVisualObject
{
public:
    pbadd();
    virtual ~pbadd();
    PBXRESULT Invoke(
        IPB_Session *session,
        pbobject obj,
        pbmethodID mid,
        PBCallInfo *ci);

    // Enum used to provide entry points for each
    // method in the class - the only one in this case
    // is mAdd
    enum MethodIDs
    {
        mAdd = 0
    };

    int f_add(IPB_Session*, pbint, pbint);
};
```

```cpp
#include "pbadd.h"

class pbadd: public IPBX_NonVisualObject
{
public:
    pbadd();
    virtual ~pbadd();
    PBXRESULT Invoke(
        IPB_Session *session,
        pbobject obj,
        pbmethodID mid,
        PBCallInfo *ci);

    int f_add(IPB_Session*, pbint, pbint);
};
```
private:
   virtual void Destroy();
};

2. Create the main.cpp file, which includes pbadd.h and implements the standard functions, PBX_GetDescription and PBX_CreateNonvisualObject:

- PBX_GetDescription is used to pass the descriptions of classes in the extension to PowerBuilder.
- The PBX_CreateNonVisualObject method creates the object instance. The PowerScript CREATE statement maps to this PNI method.

The following is the code for main.cpp:

```c++
#include "pbadd.h"
// initialize the PBX
BOOL APIENTRY DllMain(HANDLE hModule,
   DWORD ul_reason_for_all,
   LPVOID lpReserved
)
{
   switch(ul_reason_for_all)
   {
      case DLL_PROCESS_ATTACH:
      case DLL_THREAD_ATTACH:
      case DLL_THREAD_DETACH:
      case DLL_PROCESS_DETACH:
         break;
   }
   return TRUE;
}

// describe the pbadd class
PBXEXPRT LPCTSTR PBXCALL PBX_GetDescription()
{
   static const TCHAR desc[]="
      "class pbadd from nonvisualobject "
      "function int f_add(int a,int b)"n" 
      "end class "n"
   
   return desc;
}
```
Nonvisual extension example

// export the required PBX_CreateNonVisualObject
// function so that the PBVM can
// create an instance of the class
PBXEXPORT PBXRESULT PBXCALL
PBX_CreateNonVisualObject
{
  IPB_Session* pbSession,
pbobject pbobj,
LPCSTR xtraName,
IPBX_NonVisualObject **obj
}
{
  // if the calling function requests the pbadd
  // class, create an instance
  if (strcmp(xtraName,"pbadd")==0)
  {
    *obj=new pbadd;
  }
  return 0;
};

3 Create the pbadd.cpp file, which includes pbadd.h and contains the implementation of the pbadd class and its single method, f_add.

#include "pbadd.h"

// Implement the required Invoke method
PBXRESULT pbadd:: Invoke(IPB_Session *Session,
pbobject obj, pbmethodID mid, PBCallInfo *ci)
{
  // if the method to call is f_add
  if (mid == mAdd)
  {
    int sum = f_add(Session, ci->pArgs->GetAt(0)->GetInt(), ci->pArgs->GetAt(1)->GetInt());
    ci->returnValue->SetInt(sum);
  }
  return PBX_OK;
}

// constructor and destructor
pbadd:: pbadd()
{
}

pbadd:: ~pbadd()
{ }
CHAPTER 2  Building PowerBuilder Extensions

// implement the class’s f_add method
int pbadd::f_add(IPB_Session* session, pbint arg1,
    pbint arg2)
{
    return arg1+arg2;
}

// Implement the required Destroy method
void pbadd::Destroy()
{
    delete this;
}

❖ To compile and link the PBX:
- In your C++ development tool or on the command line, compile and link
  the PBX.

Make sure the include directory in PowerBuilder 12.5\SDK\PBNI is in
your include path. For this example, the generated DLL is called
pbadd.pbx.

Using the extension in PowerBuilder

To use the PowerBuilder native class in a PowerBuilder application, import the
object descriptions in the PBX file into a library in your application.

❖ To import the extension into an application:
1 Copy the PBX (or DLL) file to a directory on your application’s path.
2 In PowerBuilder, create a new workspace.
3 On the Target page of the New dialog box, select the Application icon to
  create a new target, library, and application object.
Nonvisual extension example

4 In the System Tree, expand the new target, right-click the library, and select Import PB Extension from the pop-up menu.

5 Navigate to the location of the pbadd.pbx file and click Open.

❖ To invoke the f_add function in PowerBuilder:

1 Create a new window called w_add, and add three single-line edit boxes and a command button to it.

2 Declare an instance variable called mypbadd for the pbadd native class, and then add this script to the button’s Clicked event:

```
TRY
  mypbadd = CREATE pbadd
  sle_3.text = string (mypbadd.f_add( &
                        integer(sle_1.text), integer(sle_2.text)))
CATCH (runtimeerror re)
  MessageBox("Error", &
             "pbadd native class could not be created: " + &
             re.getmessage() )
END TRY
```

The pbadd class displays in the System Tree. As shown in the following screen shot, you can expand its function list:
3 Add open(w_add) to the application’s Open event.

4 Run the application.

The application runs just as it would if you had created a custom class user object in PowerBuilder with an f_add function. If PowerBuilder cannot find pbadd.pbx, the runtime error in the Clicked event script will be triggered and caught. Put pbadd.pbx in the same directory as the executable or the PowerBuilder runtime DLLs to make sure it can be found.

Creating a PowerBuilder extension

To build a PowerBuilder extension, follow these steps:

Step 1: Decide on a feature to implement.

Step 2: Define the classes and functions in the extension.

Step 3: Declare native classes and global functions.

Step 4: Implement native classes and global functions.

Step 5: Export methods to create class instances.

Step 6: Build a PBX.

These steps apply whether you are building a nonvisual or a visual extension. The differences between building nonvisual and visual extensions are described in “Creating and using a visual extension” on page 28. This section focuses primarily on nonvisual extensions.

Required methods

All PowerBuilder nonvisual extensions must export two methods: PBX_GetDescription and PBX_CreateNonVisualObject. The use of these methods is described in “Step 2: Define the classes and functions in the extension” on page 18 and “Step 5: Export methods to create class instances” on page 24.

PowerBuilder visual extensions must export PBX_GetDescription and PBX_CreateVisualObject. See “Creating and using a visual extension” on page 28.

If the extension declares global functions, it must also export the PBX_InvokeGlobalFunction method.
Creating a PowerBuilder extension

For every native class, you must implement two PNI methods, Invoke and Destroy, in addition to the methods the class will provide. The use of these PNI methods is described in “Step 4: Implement native classes and global functions” on page 22.

Step 1: Decide on a feature to implement

The first step in building a PowerBuilder extension is to identify a problem that an extension can solve. This might be a feature that can be coded more efficiently and easily in C++ than in PowerScript, or that requires the use of callback functions or nonstandard datatypes. You might also have access to existing C++ classes that perform the tasks you want to add to a PowerBuilder application, or you might want to create a wrapper for existing standard utilities written in C++.

For possible uses of PowerBuilder extensions, see “Understanding PowerBuilder extensions” on page 4.


Step 2: Define the classes and functions in the extension

Your C++ code must expose two standard methods that enable PowerBuilder to recognize each native class and create instances of the class. One of these methods is PBX_GetDescription.

Use PBX_GetDescription to pass the descriptions of classes and global functions in the PowerBuilder extension to PowerBuilder. Every extension must export this method. Importing the PBX or DLL file into a PBL converts the description of the extension into PowerScript and adds it to the PBL as source code. The keyword native in the source code indicates that the PowerBuilder type was defined in an extension.

All the classes or global functions in an extension module are passed in a single description. The examples that follow illustrate how you define classes and functions in a description. For the full syntax, see PBX_GetDescription on page 246.
Describing nonvisual classes

Nonvisual classes can inherit from the NonVisualObject PowerBuilder system class or any of its descendants. While a native class can inherit from a user-defined user object, Sybase recommends that you use only system classes. Each native class can provide several functions, subroutines, and events.

The following example shows how you use the PBX_GetDescription method in the C++ code for an extension that includes three nonvisual classes. ClassName1 inherits from NonVisualObject, ClassName2 inherits from Exception, and ClassName3 inherits from Transaction. All three classes must be in a single description passed by PBX_GetDescription:

```cpp
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[] = {
        // Description begins here
        "class ClassName1 from NonVisualObject\n"
        "function integer objectFunction(integer a[])\n"
        "subroutine objectSubroutine(integer ai_ref)\n"
        "event integer eventName(integer b)\n"
        "end class\n"

        "class ClassName2 from Exception\n"
        "function integer objectFunction(readonly integer ai)\n"
        "subroutine objectSubroutine(integer arg)\n"
        "event integer eventName(integer arg)\n"
        "end class\n"

        "class ClassName3 from Transaction\n"
        "function integer objectFunction(integer arg)\n"
        "subroutine objectSubroutine(integer arg)\n"
        "event integer eventName(integer arg)\n"
        "end class\n"
        // Description ends here
    }
    return desc;
}
```

Describing visual classes

Visual native classes can inherit only from the UserObject PowerBuilder system class. The PowerBuilder VM considers any class that inherits from UserObject to be a visual class. All other native classes are considered to be nonvisual classes. For more information about how to describe visual classes, see “Creating and using a visual extension” on page 28.
Creating a PowerBuilder extension

Describing global functions

An extension can include global functions as well as classes. This example shows a description for two global functions:

```
"globalfunctions \n" 
"function int g_1(int a, int b)\n" 
"function long g_2(long a, long b)\n" 
"end globalfunctions\n"
```

The syntax and usage of global functions defined in an extension are the same as for global functions defined in the Function painter in PowerBuilder.

Global functions cannot be overloaded

Like global functions in PowerScript, global functions in a PowerBuilder extension cannot be overloaded.

Using forward declarations

PowerBuilder extensions can provide multiple classes. A class can reference any class that is defined earlier in the description, but if it references a class defined later in the description, you must provide a forward declaration. This example shows a description that includes forward declarations for two classes, nativeclass_1 and nativeclass_2, that reference each other. This example also demonstrates that a single description can include global functions as well as classes:

```
"forward\n" 
"class nativeclass_1 from nonvisualobject\n" 
"class nativeclass_2 from nonvisualobject\n" 
"end forward\n" 

"class nativeclass_1 from nonvisualobject \n" 
"function int add(nativeclass_2 a, int b)\n" 
"function int sub(int a, int b)\n" 
"end class \n"

"class nativeclass_2 from nonvisualobject \n" 
"function int add(nativeclass_1 a, int b)\n" 
"function int sub(int a, int b)\n" 
"end class \n"

"globalfunctions \n" 
"function int g_1(int a, int b)\n" 
"function long g_2(long a, long b)\n" 
"end globalfunctions\n"
```
Step 3: Declare native classes and global functions

For each native class that the nonvisual extension supports, declare an ANSI C++ class that inherits from IPBX_NonVisualObject, which is the ancestor class for all nonvisual PowerBuilder native classes.

The declaration of the class can be placed in a header file, and it must include Invoke and Destroy methods. This is a simple prototype for a nonvisual class:

```c++
#include "pbext.h"

class CMyClass : public IPBX_NonVisualObject
{
    enum MethodIDs
    {
        mFunca = 0,
        mFuncb = 1
    };

    public:
    // constructor, destructor
    CMyClass()
    virtual ~CMyClass()

    // member methods
    PBXRESULT Invoke(
        IPB_Session *session,
        pboobject obj,
        pbmethodID mid,
        PBCallInfo *ci
    );
    void Destroy();

    private:
    void funcA(IPB_Session* session, pboobject obj,
        PBCallInfo* ci);
    void funcB(IPB_Session* session, pboobject obj,
        PBCallInfo* ci);
};
```
If you declare global functions in your extension, the extension must export the PBX_InvokeGlobalFunction method. The following PBX_GetDescription call declares three global functions: bitAnd, bitOr, and bitXor:

```c
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[] = {
        "globalfunctions\n"
        "function int bitAnd(int a, int b)\n"
        "function int bitOr(int a, int b)\n"
        "function int bitXor(int a, int b)\n"
        "end globalfunctions\n"
    };

    return desc;
}
```

**Step 4: Implement native classes and global functions**

The implementation of each class must include the implementation of the Invoke and Destroy methods, as well as all the methods declared for the class. Invoke and Destroy are methods of the IPBX_UserObject interface.

When the PowerBuilder application calls a method on the native class, the PBVM calls the Invoke method, which dispatches the call based on the method ID or method name. The method name is used when the method is called dynamically.

The Invoke method must be coded to invoke each method in the class. The example that follows shows a switch-case statement that invokes either funcA or funcB, depending on the value of the method ID. When the PowerBuilder application has finished using an instance of a native class, the PBVM calls the Destroy method.
This example does not show the implementation of the methods of the class itself:

```c++
PBXRESULT MyClass::Invoke(IPB_Session *session, pbobject obj, pbmethodID mid, PBCallInfo *ci) {
    PBXRESULT result = PBX_OK;

    switch (mid) {
    case mFunca:
        result = funcA(session, obj, ci);
        break;

    case mFuncb:
        result = funcB(session, obj, ci);
        break;

    default:
        result = PBX_E_INVOKE_FAILURE;
        break;
    }

    return result;
}
// Implementation of funcA and funcB not shown
void Destroy() {
    delete this;
}
```
Creating a PowerBuilder extension

The following PBX_InvokeGlobalFunction contains the implementation of the three global functions included in the description shown in “Step 3: Declare native classes and global functions” on page 21:

```c
PBXEXPORT PBXRESULT PBXCALL PBX_InvokeGlobalFunction
{
    IPB_Session* pbsession,
    LPCTSTR functionName,
    PBCallInfo* ci
}

PBXRESULT pbrResult = PBX_OK;

int arg1 = ci->pArgs->GetAt(0)->GetInt();
int arg2 = ci->pArgs->GetAt(1)->GetInt();

if (stricmp(functionName, "bitand") == 0)
{
    ci->returnValue->SetInt(arg1 & arg2);
}else if (stricmp(functionName, "bitor") == 0)
{
    ci->returnValue->SetInt(arg1 | arg2);
}else if (stricmp(functionName, "bitxor") == 0)
{
    ci->returnValue->SetInt(arg1 ^ arg2);
}else
{
    return PBX_FAIL;
}

return pbrResult;
```

**Step 5: Export methods to create class instances**

PowerBuilder creates nonvisual and visual class instances differently:

- For visual classes, the instance is created when the window or visual control in which the class is used is opened. See “Creating visual class instances” on page 31.
- For nonvisual classes, the instance is created when the PowerBuilder CREATE statement is used. This is described next.
When the PowerBuilder application creates an instance of a nonvisual class using the PowerScript CREATE statement, the PBVM calls the PBX_CreateNonVisualObject method in the extension. Every extension that contains nonvisual native classes must export this method.

In the same way that multiple classes are included in a single description passed by PBX_GetDescription, PBX_CreateNonVisualObject can be used to create multiple classes.

In this example, the extension has three classes. An IF statement compares the name of the class passed in from the PowerBuilder CREATE statement to the name of each of the classes in the extension in turn and creates an instance of the first class with a matching name. You could also use a CASE statement. The class name in the string comparison must be all lowercase:

```c
PBXEXPORT PBXRESULT PBXCALL PBX_CreateNonVisualObject(  
    IPB_Session * session,  
    pbobject obj,  
    LPCSTR className,  
    IPBX_NonVisualObject **nvobj
)
{
    PBXRESULT result = PBX_OK;
    // The class name must not contain uppercase
    if ( strcmp( className, "classone" ) == 0 )
        *nvobj = new ClassOne;
    else if ( strcmp( className, "classtwo" ) == 0 )
        *nvobj = new ClassTwo( session );
    else if ( strcmp( className, "classthree" ) == 0 )
        *nvobj = new ClassThree;
    else
    {
        *nvobj = NULL;
        result = PBX_E_NO_SUCH_CLASS;
    }
    return PBX_OK;
};
```
Step 6: Build a PBX

Using your C++ development tool or the command line, build a PBX from your C++ classes.

When you compile and link the C++ code, verify the following:

- The include directory for the PBNI SDK, typically PowerBuilder 12.5\SDK\PBNInclude, must be in your include path.

- If you use any helper classes, make sure the source file that contains them is added to your project. For a list of classes, see the table in “The PBNI SDK” on page 8.

Now you are ready to use the extension in a PowerBuilder application.

Adding an extension to a PowerBuilder target

The simplest way to add a PowerBuilder native class to a PowerBuilder target is to import the object descriptions in the PBX file into a library in the PowerBuilder System Tree. You can also use a command-line tool to create a PBD file from a PBX file and add it to the target’s library search path. See pbx2pbd125 on page 259.

❖ To import the descriptions in an extension into a library:
1. Copy the PBX file into a directory on the application’s path.
2. In the System Tree, expand the target in which you want to use the extension, right-click a library, and select Import PB Extension from the pop-up menu.
3. Navigate to the location of the PBX file and click Open.

Each class in the PBX displays in the System Tree so that you can expand it, view its properties, events, and methods, and drag and drop to add them to your scripts.

Using the extension

Using nonvisual classes

In PowerScript, use the classes in a nonvisual extension just as you would a custom class user object: Declare an instance of the object, use the `CREATE` statement to create the instance, invoke the object’s functions, and destroy the instance when you have finished with it. You can inherit from the native classes if you want to add functions or events to the class.

At runtime, instances of the native class are created as normal PowerBuilder objects.

In this example, the extension module contains two nonvisual native classes: `fontcallback` and `fontenumerator`. A PowerBuilder custom class user object, `nvo_font`, inherits from the `fontcallback` class. These statements create instances of both classes:

```powerbuilder
fontenumerator fe
nvo_font uf
fe = create fontenumerator
uf = create nvo_font
```

After an instance of a native class has been created, the PowerBuilder application can call methods on the object. Each native class must implement an `Invoke` method that the PowerBuilder VM calls when the PowerBuilder application calls one of the native class’s methods. Then, the `Invoke` method dispatches the method call based on the method ID or method name. The method name is used when a native method is called dynamically.

Using the previous example, this statement invokes the `enumprinterfonts` method of the instance of the `fontenumerator` class:

```powerbuilder
fe.enumprinterfonts(uf)
```

Destroying the PBNI object instance

When the PowerBuilder application no longer needs an instance of a nonvisual class and a `DESTROY` statement is issued, by either the user or the garbage collector, or when the window or visual control that contains a visual class is closed, the PowerBuilder VM destroys the instance by calling the native class’s `Destroy` method.
Creating and using a visual extension

In general, you follow the same steps to create and use a visual extension that you do to create a nonvisual extension:

Step 1: Decide on a feature to implement.
Step 2: Define the classes and functions in the extension.
Step 3: Declare visual classes and global functions.
Step 4: Implement native classes.
Step 5: Export methods to create class instances.
Step 6: Build and use a PBX.
Step 7: Use the visual extension in an application.

Using PowerBuilder visual objects in C++
For information about using PowerBuilder visual objects from a C++ application, see “Processing PowerBuilder messages in C++” on page 73.

Step 1: Decide on a feature to implement

You can choose to use visual extensions to implement controls with a specific purpose or that use a custom look and feel. For some examples of visual extensions, see the PowerBuilder CodeXchange Web site at http://powerbuilder.codexchange.sybase.com.
Step 2: Define the classes and functions in the extension

The description for a visual class follows the same rules as for a nonvisual class, but it must inherit from the UserObject system class:

```cpp
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[] = {
        "class myvisualext from userobject\n"
        "subroutine func_1(int arg1, int arg2)\n"
        "subroutine func_2(string arga)\n"
        "end class\n"
    };

    return desc;
}
```

There are no events in the preceding example, but a typical visual extension makes use of events such as mouse clicks. There are two ways to declare and handle events. See “Event processing in visual extensions” on page 34.

Step 3: Declare visual classes and global functions

You declare native visual classes in the same way as nonvisual classes, except that you declare an ANSI C++ class that inherits from IPBX_VisualObject, which is the ancestor class for all nonvisual PowerBuilder native classes, instead of from IPBX_NonVisualObject. You can also declare global functions in a visual extension. See “Step 3: Declare native classes and global functions” on page 21 in the section on nonvisual extensions.

Step 4: Implement native classes

You implement Invoke and Destroy methods and any class or global functions the same way for visual extensions as for nonvisual extensions. See “Step 4: Implement native classes and global functions” on page 22.

Step 5: Export methods to create class instances

The major difference between visual and nonvisual extensions is in how instances of the class are created. See “Creating visual class instances” on page 31.
Creating and using a visual extension

Step 6: Build and use a PBX

As for nonvisual extensions, you must build a PBX, import it into the application, and put the PBX in the execution path. See “Step 6: Build a PBX” on page 26 and “Adding an extension to a PowerBuilder target” on page 26 in the section on nonvisual extensions.

Step 7: Use the visual extension in an application

You do not need to declare an instance of a visual class or use the CREATE statement to create an instance. The PBVM creates an instance when the window or visual control in which the visual class resides is opened, as described in “Creating visual class instances” on page 31. You can invoke the object’s functions the same way that you invoke a nonvisual object’s functions.

❖ To use a visual extension:

1. Select File>Inherit from the PowerBuilder menu and select the PBD in the Libraries list in the Inherit from Object dialog box.
2. Select the visual class and click OK.
3. In the User Object painter, size the visual object and make any other changes you need.
4. Save the object.

You can now drag the new user object from the System Tree directly onto a window or onto another visual control, such as a tab control, and use it like any other visual user object.

Code samples

The code fragments in the rest of this section are based on complete sample applications that you can find on the PowerBuilder CodeXchange Web site at http://powerbuilder.codexchange.sybase.com.
Creating visual class instances

When the window or other visual control in which a visual native class resides is created in a PowerBuilder application, the PBVM calls the PBX_CreateVisualObject method in the extension automatically—the PowerBuilder application developer does not need to write a CREATE statement in a script. The PBVM also calls the IPBX_VisualObject’s CreateControl method. Every extension that contains visual native classes must export PBX_CreateVisualObject and implement CreateControl.

The following is sample code for PBX_CreateVisualObject:

```c
PBXEXPORT PBXRESULT PBXCALL PBX_CreateVisualObject
{
    IPB_Session* pbsession,
    pbobject pobj,
    LPCTSTR className,
    IPBX_VisualObject **obj
}

    PBXRESULT result = PBX_OK;

    string cn(className);
    if (cn.compare("visualext") == 0)
    {
        *obj = new CVisualExt(pbsession, pbobj);
    }
    else
    {
        *obj = NULL;
        result = PBX_FAIL;
    }

    return PBX_OK;
};
```
Creating visual class instances

Registering the window class

Before 

CreateControl 
can be called, the window class must be registered. This code uses the Windows RegisterClass method to register the window class with the class name "s_className":

```cpp
void CVirtualExt::RegisterClass()
{
    WNDCLASS wndclass;

    wndclass.style = CS_GLOBALCLASS | CS_DBLCLKS;
    wndclass.lpfnWndProc = WindowProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = g_dll_hModule;
    wndclass.hIcon = NULL;
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = s_className;

    ::RegisterClass (&wndclass);
}
```

You must also implement the Windows UnregisterClass method to unregister the class when the window is closed:

```cpp
void CVirtualExt::UnregisterClass()
{
    ::UnregisterClass(s_className, g_dll_hModule);
}
```

The RegisterClass and UnregisterClass methods are called in the initialization code for the PBX. This is the Visual C++ DllMain method:

```cpp
BOOL APIENTRY DllMain(HANDLE hModule,
DWORD  reasonForCall,
LPVOID lpReserved
)
{
    g_dll_hModule = (HMODULE)hModule;

    switch (reasonForCall)
    {
    case DLL_PROCESS_ATTACH:
        CVirtualExt::RegisterClass();
        break;

    case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
```
break;

case DLL_PROCESS_DETACH:
    CVisualExt::UnregisterClass();
    break;
}
return TRUE;
}

Every visual native class must implement the IPBX_VisualObject CreateControl method. After getting the class name with the IPBX_VisualObject GetClassName method, CreateControl passes the class name to the Windows CreateWindowEx method to create the window, then returns the window handle to the PBVM:

TCHAR CVisualExt::s_className[] = "PBVisualExt";

LPCTSTR CVisualExt::GetWindowClassName()
{
    return s_className;
}

HWND CVisualExt::CreateControl
(
    DWORD dwExStyle, // extended window style
    LPCTSTR lpWindowName, // window name
    DWORD dwStyle, // window style
    int x, // horizontal position of window
    int y, // vertical position of window
    int nWidth, // window width
    int nHeight, // window height
    HWND hWndParent, // handle to parent or
    HINSTANCE hInstance // handle to application
)
{
    d_hwnd = CreateWindowEx(dwExStyle, s_className,
                            lpWindowName, dwStyle, x, y, nWidth, nHeight,
                            hWndParent, NULL, hInstance, NULL);

    ::SetWindowLong(d_hwnd, GWL_USERDATA, (LONG)this);
    return d_hwnd;
}
Event processing in visual extensions

A visual extension can have a window procedure that can process any Windows message or user-defined message. The PBVM passes all such messages to the visual extension, which can intercept messages and either process or ignore them.

WindowProc is an application-defined callback function that processes messages sent to a window. In the example in this section, a WM_PAINT message is sent to the extension when an action in the PowerBuilder application causes the window to be redrawn. When the extension receives the message, it repaints an area in the window using the current values for text and color set by the user of the application.

The following example also captures mouse clicks and double clicks, and triggers the Onclick and Ondoubleclick event scripts in the PowerBuilder application. You can use two different syntaxes for describing events:

1. **event returnType eventName(args_desc) newline**
2. **event eventName pbevent_token newline**

Using an event name with return type and arguments

The following description uses the first syntax. The class has two events, onclick and ondoubleclick:

```c
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
  static const TCHAR desc[] = {
    "class visualext from userobject\n    "event int onclick()\n    "event int ondoubleclick()\n    "subroutine setcolor(int r, int g, int b)\n    "subroutine settext(string txt)\n    "end class\n  };
  return desc;
}
```
Capturing messages and mouse clicks

The code in the extension captures the Windows messages that cause the window to be drawn, as well as mouse clicks and double clicks:

```c
LRESULT CALLBACK CVisualExt::WindowProc(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
)
{
    CVisualExt* ext = (CVisualExt*)::GetWindowLong(hwnd,
        GWL_USERDATA);
    switch(uMsg) {
        case WM_CREATE:
            return 0;
        case WM_SIZE:
            return 0;
        case WM_COMMAND:
            return 0;
        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);
            RECT rc;
            GetClientRect(hwnd, &rc);
            LOGBRUSH lb;
            lb.lbStyle = BS_SOLID;
            // Get color using the visual class’s GetColor method
            lb.lbColor = ext->GetColor();
            HBRUSH hbrush = CreateBrushIndirect(&lb);
            HBRUSH hbrOld = (HBRUSH)SelectObject(hdc, hbrush);
            Rectangle(hdc, rc.left, rc.top, rc.right-rc.left,
                rc.bottom-rc.top);
            SelectObject(hdc, hbrOld);
            DeleteObject(hbrush);

            // Get text using the visual class’s GetText method
            DrawText(hdc, ext->GetText(),
                ext->GetTextLength(), &rc,
                DT_CENTER|DT_VCENTER|DT_SINGLELINE);
            EndPaint(hwnd, &ps);
        }
    }
}
```
Event processing in visual extensions

```cpp
    return 0;

    // Trigger event scripts in the PowerBuilder application
    case WM_LBUTTONDOWN:
        ext->TriggerEvent("onclick");
        break;
    case WM_LBUTTONDBLCLK:
        ext->TriggerEvent("ondoubleclick");
        break;
    }
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
```

**Triggering click events**

The following is the `TriggerEvent` method that triggers the `onclick` and `ondoubleclick` events:

```cpp
void CVisualExt::TriggerEvent(LPCTSTR eventName)
{
    pbclass clz = d_session->GetClass(d_pbobj);
    pbmethodID mid = d_session->GetMethodID(clz,
        eventName, PBRT_EVENT, "I");

    PBCallInfo ci;
    d_session->InitCallInfo(clz, mid, &ci);
    d_session->TriggerEvent(d_pbobj, mid, &ci);
    d_session->FreeCallInfo(&ci);
}
Using an event name with a PowerBuilder event ID

A simpler way to trigger events in a visual extension uses direct mapping of Windows messages to PowerBuilder events. The following class description contains the same two events, but in this case they use the alternative syntax that maps the event name to a PowerBuilder token name:

```cpp
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[] = {
        "class visualext from userobject\n"
        "event onclick pbm_lbuttonup\n"
        "event ondoubleclick pbm_lbuttondblclk\n"
        "subroutine setcolor(int r, int g, int b)\n"
        "subroutine settext(string txt)\n"
        "end class\n"
    };
    return desc;
}
```

Generating event syntax automatically

Importing the extension generates the Onclick and Ondoubleclick events with the appropriate arguments automatically, and at runtime, the PBVM fires the events. You do not need to capture the Windows messages WM_LBUTTONUP and WM_LBUTTONDOWN in the extension.

In the following description, `onclick` is the event name and `pbm_lbuttonup` is the event token name. Notice that the event name is not followed by empty parentheses as it is when you use the return type and arguments technique for describing the event:

"event onclick pbm_lbuttonup\n"

About the token name

The token name is a string that maps to an internal PowerBuilder event ID defined in the header file `pbevtid.h`. The first ID in this file is `PB_NULL`. For all other IDs in the file, there is a fixed relationship between the name that you use in the description and the event ID in `pbevtid.h`. The name is the same as the ID with the letter `m` appended to the `pb` prefix. You must use lowercase in the description.

For example, the event ID `PB_ACTIVATE` in `pbevtid.h` maps to the token name `pbm_activate`. In the description provided with `PBX_GetDescription`, you must use the name `pbm_activate`. If the event name you provide does not exist, importing the extension generates an error message. See the `pbevtid.h` file for a complete list of mapped IDs.
Event processing in visual extensions

Processing events sent to the parent of the window

Some Windows messages, such as WM_COMMAND and WM_NOTIFY, are sent to the parent of an object and not to the object itself. Such messages cannot be caught in the visual extension’s window procedure. The PBVM calls the GetEventID method to process these messages, as follows:

- If the message is mapped to a PowerBuilder event, GetEventID returns the event’s identifier, for example PB_BNCLICKED, and the event is fired automatically.
- If the message is not mapped to an event, GetEventID returns the value PB_NULL and the message is discarded.

In the following example, the GetEventID function returns the identifier PB_BNCLICKED if a WM_COMMAND message with the notification code BN_CLICKED was sent. It returns the identifier PB_ENCHANGE if a WM_NOTIFY message was sent. Otherwise, it returns PB_NULL.

```cpp
TCHAR CVisualExt::s_className[] = "PBVisualExt";

LPCTSTR CVisualExt::GetWindowClassName()
{
    return s_className;
}

HWND CVisualExt::CreateControl(
    DWORD dwExStyle, // extended window style
    LPCTSTR lpWindowName, // window name
    DWORD dwStyle, // window style
    int x, // horizontal position of window
    int y, // vertical position of window
    int nWidth, // window width
    int nHeight, // window height
    HWND hWndParent, // handle of parent or owner window
    HINSTANCE hInstance // handle of application instance
)
{
    d_hwnd = CreateWindowEx(dwExStyle, s_className,
        lpWindowName, dwStyle, x, y, nWidth, nHeight,
        hWndParent, NULL, hInstance, NULL);

    ::SetWindowLong(d_hwnd, GWL_USERDATA, (LONG)this);

    return d_hwnd;
}
```
int CVisualExt::GetEventID(  
    HWND hWnd,  /* Handle of parent window */  
    UINT iMsg,  /* Message sent to parent window*/  
    WPARAM wParam,  /* Word parameter of message*/  
    LPARAM lParam /* Long parameter of message*/  
)  
{  
    if (iMsg == WM_COMMAND)  
    {  
        if ((HWND)lParam == d_hwnd)  
        {  
            switch(HIWORD(wParam))  
            {  
                case BN_CLICKED:  
                    return PB_BNCLICKED;  
                    break;  
            }  
        }  
    }  
    if (iMsg == WM_NOTIFY)  
    {  
        return PB_ENCHANGE;  
    }  
    return PB_NULL;  
}

Calling PowerScript from an extension

You can call PowerBuilder system functions through IPB_Session. The InitCallInfo method simplifies the process of setting up the call information. You need to provide the arguments to the InitCallInfo method, including an identifier for the PowerBuilder function you want to call.

The identifier can be returned from theGetMethodID or FindMatchingFunction method.

Using GetMethodID

To get the function’s ID using theGetMethodID method, you need the function’s signature:

```
PbmethodID GetMethodID(pbclass cls, LPCTSTR
methodName, PBRoutineType rt, LPCTSTR signature);
```
Calling PowerScript from an extension

The *signature* argument in this method call is a string representing the method’s return type and arguments. You can obtain this string in the Browser.

For example, to obtain the signature of a system function, select `systemfunctions` from the left pane of the System page, right-click the function in the right pane, and select Properties from its pop-up menu:

![Browser window showing signature of a system function](image)

For methods in your application, you can expand the object that contains it in the System Tree, select the function or event, and select Properties from its pop-up menu:

![Browser window showing signature of an application method](image)

Consider this function:

```powerbuilder
of_get_trans ( ref transaction atr_trans ) returns (none)
```
CHAPTER 2  Building PowerBuilder Extensions

The signature for this function is QRCtransaction. Q indicates that the function does not return a value, R that the argument is passed by reference, and Ctransaction that the argument is a PowerBuilder system object of type transaction.

You can use the pbsig125 command-line tool to obtain a function’s signature. However, the pbsig125 tool does not report the signature of functions that are inherited from an ancestor object unless they are extended in the descendant, and it does not report event signatures.

For more information about using pbsig125, and an explanation of all the formats used in the signature, see pbsig125 on page 255.

Instead of the string that GetMethodID uses, the FindMatchingFunction function provides another way to get the method ID. Some short signatures can be difficult to parse, and signatures that include PowerBuilder system objects or Java classes can be much longer.

FindMatchingFunction uses a “readable signature” instead of the string used by GetMethodID:

\[
\text{FindMatchingFunction}(\text{pbclass } cls, \text{ LPCTSTR methodName, PBRoutineType rt, LPCTSTR readableSignature})
\]

The readableSignature argument is a comma-separated list of the arguments of the function. Unlike the string used by GetMethodID, it does not include the return type. For example, for a function called uf_test that takes two arguments, an int by value and a double by reference, the call to FindMatchingFunction looks like this:

\[
\text{mid} = \text{Session} \to \text{FindMatchingFunction}(cls, "uf_test", \text{PBRT_FUNCTION, } \text{"int, double"});
\]

The following methods are those you use most frequently to invoke PowerBuilder functions. For descriptions of each method, see IPB_Session interface on page 96.

\[
\begin{align*}
\text{PbmethodID GetMethodID}(\text{pbclass } cls, \text{ LPCTSTR methodName, PBRoutineType rt, LPCTSTR signature, pbboolean publiconly}) \\
\text{PBXRESULT InitCallInfo}(\text{pbclass } cls, \text{ pbmethodID mid, PBCallInfo *ci}) \\
\text{void FreeCallInfo}(\text{PBCallInfo *ci}) \\
\text{PBXRESULT Add<Type>Argument}(\text{PBCallInfo *ci, PBTType v}; \\
\text{PBXRESULT InvokeClassFunction}(\text{pbclass } cls, \text{ pbmethodID mid, PBCallInfo *ci})
\end{align*}
\]
Calling PowerScript from an extension

PBXRESULT InvokeObjectFunction(pbobject obj, pbmethodID mid, PBCallInfo *ci)
PBXRESULT TriggerEvent(pbobject obj, pbmethodID mid, PBCallInfo *ci)

Example: Calling PowerBuilder functions

In this code fragment, the class and method ID returned by calls to the IPB_Session GetClass and GetMethodID methods are used to initialize a PBCallInfo structure, called ci, using the IPB_Session InitCallInfo method.

After a new pbstring variable is created, the value of that string is set to the value of the first argument in the PBCallInfo structure.

```c
BOOL CALLBACK CFontEnumerator::EnumFontProc
{
    LPLOGFONT lplf,
    LPNEWTEXTMETRIC lpntm,
    DWORD FontType,
    LPVOID userData

    UserData* ud = (UserData*)userData;
    pbclass clz = ud->session->GetClass(ud->object);
    pbmethodID mid = ud->session->GetMethodID
    (clz, "onnewfont", PBRT_EVENT, "IS");

    PBCallInfo ci;
    ud->session->InitCallInfo(clz, mid, &ci);

    // create a new string variable and set its value
    // to the value in the first argument in the
    // PBCallInfo structure
    pbstring str = ud->session->NewString
    (lplf->lfFaceName);
    ci.pArgs->GetAt(0)->SetString(str);

    ud->session->TriggerEvent(ud->object, mid, &ci);
    pbint ret = ci.returnValue->GetInt();
    ud->session->FreeCallInfo(&ci);
    return ret == 1 ? TRUE : FALSE;
}
```
Exception handling and debugging

To handle errors, you use the error codes returned from PBNI methods. Some functions of the IPB_Session interface return detailed error codes that make debugging easier.

Native methods, such as the IPBX_UserObject Invoke method, return either PBX_OK or PBX_FAIL if the extension encounters a serious problem from which it cannot recover.

Whenever the PowerBuilder VM gets PBX_FAIL from a native method, it throws a PBXRuntimeError in the PowerBuilder application. PBXRuntimeError inherits from the PowerBuilder RuntimeError system object and can be caught and handled in a script in the same way as any exception in PowerBuilder.

To catch these errors, wrap your PowerScript call in a try-catch block as follows:

```powerscript
TRY
    n_cpp_pbniobj   obj
    obj = CREATE n_cpp_pbniobj
    obj.of_test( arg1 )
CATCH ( PBXRuntimeError re )
    MessageBox( "Caught error", re.getMessage() )
END TRY
```

The IPB_Session interface provides a set of methods to handle exceptions that occur in native code. Use HasExceptionThrown to determine whether an exception occurred. If it did, use GetException to get the current exception object so that it can be handled. If necessary, you can throw exceptions to PowerBuilder with ThrowException. When an exception has been handled, use ClearException to clear it.

Debugging

You cannot edit a native class in the PowerBuilder development environment, and you cannot enter native methods in the PowerBuilder debugger because the methods are C++ methods. You must use a C/C++ debugger to debug an extension module.
Exception handling and debugging
Creating Marshaler Extensions

About this chapter

This chapter describes how to create marshaler extensions.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About marshaler extensions</td>
<td>45</td>
</tr>
<tr>
<td>Developing the PowerBuilder extension</td>
<td>46</td>
</tr>
<tr>
<td>Generating proxies for Java classes</td>
<td>53</td>
</tr>
<tr>
<td>Calling the Java class from PowerBuilder</td>
<td>54</td>
</tr>
</tbody>
</table>

About marshaler extensions

Marshaler extensions can act as bridges between PowerBuilder and other components, such as EJB components, Java classes, and Web services, as long as those components can be called from C++.

To create a marshaler extension, build a PBX that contains at least one class that implements the IPBX_Marshaler interface, as well as one or more native classes. The extension must contain code that associates the marshaler with a proxy for the component you want to call.

If you build a marshaler extension, you should also provide a tool that generates proxies so the components can be called from PowerBuilder. For example, PowerBuilder provides a marshaler extension for calling EJB components from PowerBuilder, and it provides a tool for generating proxies for EJB components.

This chapter provides an overview based on the Java Marshaler sample application, which can be downloaded from the PowerBuilder CodeXchange Web site at http://powerbuilder.codexchange.sybase.com, and shows some extracts from the sample.
This chapter describes the major tasks involved in:

- Developing the PowerBuilder extension
- Generating proxies for Java classes
- Calling the Java class from PowerBuilder

This chapter does not show detailed code samples, and the fragments shown simplify the coding involved. For a more complete understanding of the process of building a marshaler extension, download the sample available on the Web site.

Developing the PowerBuilder extension

A PowerBuilder marshaler extension usually provides a native class that acts as a creator. This class defines a function that creates an instance of the foreign component that is specified in the parameters passed into the function (1). If it succeeds in creating an instance of the foreign component (2), it creates a proxy for it using the PBVM (3, 4), creates a marshaler object (5), and associates the marshaler with the proxy (6).

*Figure 3-1: Creating a foreign component, proxy, and marshaler*
When a function of the proxy object is called in PowerScript, the PBVM calls the InvokeRemoteMethod function on the marshaler object through the proxy (1, 2). The marshaler object translates PowerBuilder function calls into requests that the foreign component understands, sends the requests (3), waits for a response, and sends the results back to PowerBuilder (4).

Figure 3-2: Invoking a remote method

To develop the extension, you need to:

Step 1: Describe the extension

Step 2: Implement the creator class

Step 3: Implement the marshaler class

Step 1: Describe the extension

The class that implements the creator, called CJavaVM in the following example, must export the PBX_GetDescription function. It inherits from NonVisualObject and has two functions, CreateJavaObject and CreateJavaVM:

```c
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[] = {
        "class javavm from nonvisualobject\n"
        "function long createjavavm(string classpath,
            string properties[])\n"
        "function long createjavaobject(ref powerobject
            proxyobject, readonly string javaclassname, 
            readonly string proxyname)\n"
        "end class\n"
    };
    return desc;
}
```

PBNI Programmers Guide and Reference 47
Step 2: Implement the creator class

Like any nonvisual native class, the CJavaVM class must implement the *Invoke* and *Destroy* functions in addition to the class functions *CreateJavaObject* and *CreateJavaVM*.

The *CreateJavaVm* function of CJavaVM gets the classpath and properties from the PBCallInfo structure. Then it loads the Java VM by calling the *loadJavaVM* function of a wrapper class called JavaVMWrapper. The JavaVMWrapper class encapsulates the JavaVM interface provided by JNI.

The *CreateJavaObject* function creates an instance of a Java class based on the given class name, creates a PowerBuilder proxy object for the Java object, creates a JavaMarshaler object, and associates the marshaler object with the proxy object.

The following is the *CreateJavaObject* function:

```c
PBXRESULT CJavaVM::CreateJavaObject
{
    IPB_Session *session,
    pbobject obj,
    PBCallInfo *ci
}
{
    enum
    {
        kSuccessful = 0,
        kInvalidJavaClassName = -1,
        kFailedToCreateJavaClass = -2,
        kInvalidProxyName = -3,
        kFailToCreateProxy = -4
    };

    // Get java class name.
    string jclassName;

    { pbstring jcn = ci->pArgs->GetAt(1)->GetPBString();
        if (jcn == NULL)
            { ci->returnValue->SetLong(kInvalidJavaClassName);
                return PBX_OK;
            }
        jclassName = jcn;
    }
```
else
{
    jclassName = session->GetString(jcn);
}
}

// Create java object
JavaVMWrapper* jvm = JavaVMWrapper::instance();
JNIEnv* env = jvm->getEnv();

jclass jcls = env->FindClass(jclassName.c_str());
jobject jobj = NULL;

if (jcls != NULL)
{
    JLocalRef lrClz(env, jcls);
    jmethodID mid = env->GetMethodID(jcls, "<init>", "()V");
    if (mid != NULL)
    {
        jobj = env->NewObject(jcls, mid);
    }
}

// Get PowerBuilder proxy name
string proxyName;

{ pbstring pn = ci->pArgs->GetAt(2)->GetPBString();

    if (pn == NULL)
    {
        ci->returnValue->SetLong(kInvalidProxyName);
        return PBX_OK;
    }

    else
    {
        proxyName = session->GetString(pn);
    }
}
// Find proxy class
pbgroup group = session->FindGroup(proxyName.c_str(),
    pbgroup_proxy);
if (group == NULL)
    {
        ci->returnValue->SetLong(kInvalidProxyName);
        return PBX_OK;
    }

pbclass cls = session->FindClass(group,
    proxyName.c_str());
if (cls == NULL)
    {
        ci->returnValue->SetLong(kInvalidProxyName);
        return PBX_OK;
    }

// Create PowerBuilder proxy object.
pbproxyObject proxy = session->NewProxyObject(cls);
if (proxy == NULL)
    {
        ci->returnValue->SetLong(kFailToCreateProxy);
        return PBX_OK;
    }

// Create JavaMarshaler
JavaMarshaler* marshaler = new JavaMarshaler(env,
    proxy, jobject);

// Associate the JavaMarshaler with the proxy
session->SetMarshaler(proxy, marshaler);

ci->pArgs->GetAt(0)->SetObject(proxy);

return PBX_OK;
}
Step 3: Implement the marshaler class

The marshaler class must implement the `InvokeRemoteMethod` function. It also needs to provide a `Destroy` function and get the handle of the module. This is the declaration:

```cpp
#include <jni.h>
#include <pbext.h>

class JavaMarshaler : public IPBX_Marshaler
{
    jobject d_jobject;
    pbproxyObject d_pbobject;

public:
    JavaMarshaler(JNIEnv* env, pbproxyObject pbobj, jobject ejbobj);
    ~JavaMarshaler();

    virtual PBXRESULT InvokeRemoteMethod(
        IPB_Session* session, 
        pbproxyObject obj, 
        LPCSTR method_name, 
        PBCallInfo* ci
    );

    virtual pbulong GetModuleHandle();

    virtual void Destroy();
};
```

The `InvokeRemoteMethod` function calls Java functions through JNI. This is the implementation in `JavaMarshaler.cpp`:

```cpp
#include "JavaMarshaler.h"
#include "JMethod.h"
#include "JavaVMWrapper.h"

extern pbulong g_dll_hModule;

pbulong JavaMarshaler::GetModuleHandle()
{
    return g_dll_hModule;
}
```
Developing the PowerBuilder extension

```cpp
//****************************************************************************
// JavaMarshaler
//****************************************************************************
JavaMarshaler::JavaMarshaler
{
  JNIEnv* env,
  pbproxyObject pbobj,
  jobject ejbobj
}
  d_jobject(env->NewGlobalRef(ejbobj)),
  d_pbobject(pbobj)
}{
}
JavaMarshaler::~JavaMarshaler()
{
  JNIEnv* env = JavaVMWrapper::instance()->getEnv();
  if (d_jobject != NULL && env != NULL)
    env->DeleteGlobalRef(d_jobject);
}
PBXRESULT JavaMarshaler::InvokeRemoteMethod
{
  IPB_Session* session,
  pbproxyObject obj,
  LPCSTR szMethodDesc,
  PBCallInfo* ci
}
{ static char* eFailedToInvokeJavaMethod =
  "Failed to invoke the Java method."
  JNIEnv* env = JavaVMWrapper::instance()->getEnv();
  JMethod method(this, szMethodDesc);
  try
  {
    if (d_jobject != NULL)
    {
      method.invoke(session, env, d_jobject, ci);
      if (env->ExceptionCheck() == JNI_TRUE)
      {
        string error(eFailedToInvokeJavaMethod);
        error += "\n";
        // Throw exception here
```
Generating proxies for Java classes

You need to develop PowerBuilder proxies for the Java classes you want to invoke from PowerBuilder. You can develop proxies using Java reflection, from Java source code directly, or using the javap tool. For example, suppose you want to invoke this Java class:

```java
public class Converter
{
    public double dollarToYen(double dollar);
    public double yenToEuro(double yen);
}
```

The PowerBuilder proxy for this Java class could be stored in a file called `converter.srx` that looks like this:

```
$PBExportHeader$converter.srx
$PBExportComments$Proxy generated for Java class
global type Converter from nonvisualobject
end type
global Converter Converter

forward prototypes
public:
function  double dollarToYen(double ad_1) alias
for "dollarToYen,(D)D"
```

```java
return PBX_E_INVALID_ARGUMENT;
}
}
}
catch(...)
{
}

return PBX_OK;
}

void JavaMarshaler::Destroy()
{
    delete this;
}
Calling the Java class from PowerBuilder

```
function double yenToEuro(double ad_1) alias
  for "yenToEuro,(D)D"
end prototypes

Notice that both PowerBuilder proxy methods have an alias containing the Java method name and method signature. This is necessary because Java is case sensitive, but PowerBuilder is not. The extension uses the alias information is used by the extension to find the corresponding Java methods.

To add the proxy to a PowerScript target, select the library where the proxy will be stored in the System Tree, select Import from the pop-up menu, and browse to select `converter.srx`.
```

Calling the Java class from PowerBuilder

In the open event of a window, create a Java VM:

```
// instance variable: javavm i_jvm
string properties[]
i_jvm = create javavm
string classpath
i_jvm.createjavavm(classpath, properties)
```

In the clicked event of a button on the window, create an instance of the Converter class using the `CreateJavaObject` method of the JavaVM instance, and call the `conv` method on the Converter class:

```
converter conv
double yen
i_jvm.createjavaobject(conv, "Converter", "converter")
yen = conv.dollarToYen(100.0)
messagebox("Yen", string(yen))
```

When the `CreateJavaObject` method is called in PowerScript, the PBVM calls the corresponding C++ method in the extension. The C++ method creates a Java Converter object through JNI. If it is successful, the method creates an instance of the PowerBuilder Converter proxy and a JavaMarshaler object, and associates the JavaMarshaler object with the PowerBuilder proxy.

When `conv.dollarToYen(100.0)` is called in PowerScript, the PowerBuilder VM calls the `InvokeRemoteMethod` method on the JavaMarshaler object, which delegates the call to the Java Converter object through JNI and returns the result to PowerBuilder.
CHAPTER 4

Exchanging Data with PowerBuilder

About this chapter

This chapter describes how PBNI extensions exchange data with PowerBuilder.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About exchanging data with PowerBuilder</td>
<td>55</td>
</tr>
<tr>
<td>Passing values between extensions and the PBVM</td>
<td>55</td>
</tr>
<tr>
<td>Using the IPB_Session interface</td>
<td>59</td>
</tr>
<tr>
<td>Saving data from IPB_Value to a local variable</td>
<td>61</td>
</tr>
<tr>
<td>Using variables throughout a session</td>
<td>63</td>
</tr>
<tr>
<td>Handling enumerated types</td>
<td>64</td>
</tr>
</tbody>
</table>

About exchanging data with PowerBuilder

You can use the IPB_Session interface or the IPB_Value and IPB_Arguments interfaces to exchange data between PowerBuilder and PBNI. The IPB_Session interface contains many virtual functions that enable the C++ code in an extension to interact with the PBVM. The IPB_Value and IPB_Arguments interfaces contain methods that you can use to pass values between PowerBuilder and extensions.

Passing values between extensions and the PBVM

PBNI uses two interfaces, IPB_Value and IPB_Arguments, to pass PowerBuilder values between the PBVM and extension PBXs. The PBNICallInfo structure holds the data.
PBCallInfo structure

The PBCallInfo structure is used to hold data and return type information for calls between extensions and the PBVM. It has three public members:

```c
IPB_Arguments* pArgs;
IPB_Value* returnValue;
pbclass returnClass;
```

The following code initializes a PBCallInfo structure using the IPB_Session InitCallInfo method. After allocating a PBCallInfo structure called `ci`, the IPB_Session GetClass and GetMethodID methods are used to get the class and method ID of the current method. Then, these parameters are used to initialize the `ci` structure:

```c
pbclass cls;
pbmethodID mid;
PBCallInfo* ci = new PBCallInfo;

cls = Session -> GetClass(myobj);
mid = Session -> GetMethodID(cls, "myfunc",
PBRT_FUNCTION, "II");

Session -> InitCallInfo(cls, mid, ci);
```

When you have finished using a PBCallInfo structure, you must call FreeCallInfo to release the allocated memory:

```c
Session -> FreeCallInfo(ci);
delete ci;
```

The IPB_Arguments and IPB_Value interfaces have methods that enable you to pass values between the PBVM and PowerBuilder extension modules using PBCallInfo to hold the data.

IPB_Arguments interface

The IPB_Arguments interface has two methods:

- `GetCount` obtains the number of arguments in a method call.
- `GetAt` obtains the value at a specific index of the `pArgs` member of the PBCallInfo structure. For each argument, `GetAt` returns a pointer to the IPB_Value interface.
The following code fragment uses GetCount and GetAt in a FOR loop to process different argument types. The ci -> pArgs -> GetCount() statement gets the number of arguments, and ci -> pArgs -> GetAt(i) gets the value at the index i. This value is a pointer to the IPB_Value interface on which IPB_Value methods, such as IsArray and GetArray, can be called (see “IPB_Value interface” next):

```c
int i;
for (i=0; i < ci-> pArgs -> GetCount();i++)
{
    pbuint ArgsType;
    if( ci -> pArgs -> GetAt(i) -> IsArray())
        pArguments[i].array_val =
            ci -> pArgs -> GetAt(i) -> GetArray();
    continue;
}
if( ci -> pArgs -> GetAt(i) -> IsObject())
{
    if (ci -> pArgs -> GetAt(i) -> IsNull())
        pArguments[i].obj_val=0;
    else
        pArguments[i].obj_val =
            ci -> pArgs -> GetAt(i) -> GetObject();
    continue;
}
...```

**IPB_Value interface**

IPB_Value has three sets of methods: helper methods, set methods, and get methods.

**Helper methods**

The IPB_Value interface helper methods provide access to information about variables and arguments, including the value’s class and type, whether it is an array or simple type, whether it is set by reference, and whether the null flag is set. There is also a method that sets the value to null:

```c
virtual pbclass GetClass() const = 0;
virtual pbint GetType() const = 0;
virtual pbboolean IsArray() const = 0;
virtual pbboolean IsObject() const = 0;
virtual pbboolean IsByRef() const = 0;
virtual pbboolean IsNull() const = 0;
virtual PBXRESULT SetToNull() = 0;
```
The example shown in the previous section, “IPB_Arguments interface” on page 56, shows how you can use three of these methods: IsArray, IsObject, and IsNull.

This example shows how you can use the SetToNull method to set the returnValue member of the PBCallInfo structure to null:

```c++
if ( ci->pArgs->GetAt(0)->IsNull() ||
    ci->pArgs->GetAt(1)->IsNull() )
{
    // if either of the passed arguments is null,
    // return the null value
    ci->returnValue->SetToNull();
}
```

Set methods

The IPB_Value set methods set values in the PBCallInfo structure. There is a set method for each PowerBuilder datatype: SetInt, SetUint, SetLong, SetUlong, and so on. These methods automatically set the value represented by IPB_Value to not null. The syntax is:

```
virtual PBXRESULT Set<type>(<pbtype> arg);
```

For example, the SetLong method takes an argument of type pblong.

In this example, the method has two integer arguments, set to int_val1 and int_val2:

```c++
   ci-> pArgs -> GetAt(0) -> SetInt(int_val1);
   ci-> pArgs -> GetAt(1) -> SetInt(int_val2);
```

The IPB_Value set methods set the datatype of the value represented by IPB_Value to a specific type. If the original type of the value is any, you can set it to any other type. Then, because the value now has a specific type, setting it to another type later returns the error PBX_E_MISMATCHED_DATA_TYPE. If the argument is readonly, the error PBX_E_READONLY_ARGS is returned.

Get methods

The IPB_Value get methods obtain values from the PBCallInfo structure. There is a get method for each PowerBuilder datatype: GetInt, GetUint, GetLong, GetUlong, and so on. The syntax is:

```
virtual <pbtype> Get<type>();
```

For example, the GetString method returns a value of type pbstring.

The following example uses the IPB_Value GetAt method to assign the value at the first index of the pArgs member of the PBCallInfo structure to a variable of type IPB_Value* called pArg. If pArg is not null, the GetLong method sets the variable longval to the value in pArg:

```c++
PBCallInfo  *ci
...
```
pblong longval = NULL;
IPB_Value* pArg = ci-> pArgs-> GetAt(0);

if (!pArg->IsNull())
  longval = pArg -> GetLong();

If the value is null, or if you use a get method that is expected to return one
datatype when the value is a different datatype (such as using GetLong when
the datatype is pbarray), the result returned is undetermined.

The get methods can also be used with the returnValue member of PBCallInfo:

    ret_val = ci.returnValue->GetInt();
    return ret_val;

Using the IPB_Session interface

The IPB_Session interface is an abstract interface that enables the PBVM to
interact with PowerBuilder extensions and with external applications. It
defines hundreds of methods for accessing PowerScript variables, calling
PowerScript methods, handling exceptions, and setting a marshaler to convert
PowerBuilder data formats to the user’s communication protocol.

The IPB_Session interface includes several categories of methods:

- Class accessor methods are used to find PowerBuilder classes, call
  PowerBuilder methods and events, and get and set instance variables of
  PowerBuilder objects.
- Exception-handling methods communicate with the PowerBuilder
  exception handling mechanism.
- Array accessor methods create and access PowerBuilder bounded and
  unbounded arrays.
- Result set accessor methods work with result sets in DataStores and
  DataWindow controls.
- Typed data access methods create and access data of the PowerBuilder
  types string, double, decimal, blob, date, time, datetime, and so forth.
- Proxy access methods provide an interface for the implementation of new
  protocols.
- The Release method releases the IPB_Session object itself.
For a complete list of methods, see IPB_Session interface on page 96.

You use IPB_Session methods in conjunction with IPB_Value and IPB_Arguments methods.

The following code fragment shows the body of a method that tests whether a date passed to a PBNI function is handled correctly by a PowerBuilder function. It uses the IPB_Value SetToNull, SetDate, and IsNull methods to set and test the date values in the PBCallInfo structure, as well as the IPB_Session SplitDate, SetDate, and NewDate methods.

```c++
// boolean isNull[], pobject myobj,
// and pdate* d_date arguments passed in
pbclass cls;
pbmethodID mid;
PBCallInfo* ci = new PBCallInfo;
pdate ret_date;
pbint yy,mm,dd;

cls = Session->GetClass(myobj);
mid = Session->GetMethodID(cls,"uf_getdate_byref",
PBRT_FUNCTION,"YR");
Session->InitCallInfo(cls, mid, ci);

if (isNull[0])
    ci -> pArgs -> GetAt(0)->SetToNull();
else
    ci-> pArgs -> GetAt(0) ->SetDate(*d_date);

Session->InvokeObjectFunction(myobj, mid, ci);

Session->SplitDate(ci->pArgs->GetAt(0)->GetDate(),
    &yy,&mm,&dd);
Session->SetDate(*d_date,yy,mm,dd);

if (ci-> returnValue ->IsNull())
{
    ret_date = Session-> NewDate();
    Session-> SetDate(ret_val, 1900, 1, 1);
}
else
{
    ret_date = Session-> NewDate();
    Session -> SplitDate(ci-> returnValue -> GetDate(),
        &yy,&mm, &dd);
    Session -> SetDate(ret_val,yy,mm,dd);
}
```
Session -> FreeCallInfo(ci);
delete ci;
return ret_date;

Saving data from IPB_Value to a local variable

To avoid memory leaks, you must call FreeCallInfo to free the values stored in the PBCallInfo structure after using the structure. However, after making a function call, you might want to save the return value or a by reference argument value into a local variable you can use later.

There are techniques for saving values so they are still available after the call to FreeCallInfo. How you save your result into a local variable depends on whether you want to save a simple value, a pointer value, or an object value.

**Saving simple values**

Saving simple values is straightforward. When you call one of the IPB_Value Get<type> methods for a simple value, such as GetInt or GetReal, the actual data is returned. As a result, you can simply save the values of any of the following datatypes:

- pbvalue_byte
- pbvalue_int
- pbvalue_uint
- pbvalue_long
- pbvalue_ulong
- pbvalue_real
- pbvalue_double
- pbvalue_longlong
- pbvalue_boolean
- pbvalue_char

**Saving pointer values**

A pointer value does not contain data. It contains a pointer to a memory location where the data is stored. When you call one of the IPB_Value Get<type> methods for a pointer value, such as GetBlob or GetTime, it returns a pointer to memory that is also pointed to by IPB_Value.

When you call FreeCallInfo, the memory to which IPB_Value points is released and the data is deleted. Because this is the same data pointed to by the pointer returned by the Get<type> method, that pointer can no longer be used to represent the data.
Saving data from IPB_Value to a local variable

This applies to the following pointer value datatypes, as well as to the pbarray datatype:

- pbvalue_dec
- pbvalue_string
- pbvalue_blob
- pbvalue_date
- pbvalue_time
- pbvalue_datetime

If you want to save the data in a pointer value, you can use the AcquireValue, AcquireArrayItemValue, and ReleaseValue methods to acquire and release the data. These methods clone a new IPB_Value that is not freed until you call ReleaseValue and reset the existing IPB_Value pointer.

Can be used for other datatypes
You can use AcquireValue and AcquireArrayItemValue to acquire values of any datatype.

Like the Get<type> methods, AcquireValue and AcquireArrayItemValue return a pointer to the memory where the data is stored, but they also reset the IPB_Value pointer so that IPB_Value no longer points to the actual data. When you call FreeCallInfo, the data pointed to by the value acquired using AcquireValue and AcquireArrayItemValue is unaffected.

The original value is reset to zero or null, so it can no longer be used. Attempts to get or acquire the original value return zero or null until another IPB_Value is set to the value.

If the IPB_Value acquired using AcquireValue is an array, the entire array is acquired. To acquire only an element of the array, use AcquireArrayItemValue. When you have finished using the data, you must free the memory using the ReleaseValue method.

The processing that the AcquireArrayItemValue and ReleaseValue methods perform results in poor performance when handling large arrays. It is more efficient to get the type of the array and handle each type with appropriate type-specific functions.

Caution
You must call the ReleaseValue method to free the data. If you do not do so, a memory leak will occur. You must not call ReleaseValue to release a pointer that was not acquired using AcquireValue and AcquireArrayItemValue. Doing so might cause the PBVM to crash.
Saving object values

Strictly speaking, object values are also pointer values, but the PBVM handles them differently. You use the IPB_Session AddLocalRef and AddGlobalRef methods to add a reference to the object. If there is a reference to an object, it is not deleted when FreeCallInfo is called.

When you no longer need the object, call RemoveLocalRef or RemoveGlobalRef to decrease the reference count for the object. If the reference count is decreased to zero, the object is deleted automatically.

There is an important difference between AddLocalRef and AddGlobalRef. A reference added by AddLocalRef can be deleted automatically when the local frame is popped up. The local frame can be popped by calling PopLocalFrame or when the current function returns. However, a reference added by AddGlobalRef is deleted only when RemoveGlobalRef is called or the session ends.

You must use these methods in pairs; that is, use RemoveLocalRef to remove references created with AddLocalRef, and use RemoveGlobalRef to remove references created with AddGlobalRef.

Using variables throughout a session

The SetProp function enables you to use a variable value throughout an IPB session without using a global variable, which is susceptible to namespace conflicts with other sessions. SetProp is one of a set of three functions:

- Use SetProp to register a new variable with the session or to change the value of an existing variable.
- Use GetProp to access the variable.
- Use RemoveProp to remove the variable from the list of variables associated with the session when it is no longer needed.

This set of functions is particularly useful for working with multiple threads of execution in EAServer.

Suppose you want to throw an exception from within a PBNI extension and the exception itself is also defined by the PBNI extension. You call the IPB_Session NewObject function to create an instance of the exception, causing the PBX_CreateNonVisualObject function to be called.
Handling enumerated types

One way to set the value of the fields of the exception before the function returns in a thread-safe manner is to create a new object or structure to hold the exception information before calling NewObject. You can call SetProp to store the structure or the object in the current IPB_Session. When PBX_CreateNonVisualObject is called, you can call GetProp to get the structure or object to obtain the exception information, then call RemoveProp to remove the data you stored in the current session.

You can also use these functions when initializing and uninitializing a session. If the extension exports the PBX_NOTIFY function, the PBVM calls PBX_Notify immediately after an extension PBX is loaded and just before the PBX is unloaded. You can use this function to initialize and uninitialize a session. For example, you could create a session manager object, and store it in the IPB session using the SetProp function. Later, you could use GetProp to obtain the session object.

Handling enumerated types

The GetEnumItemValue and GetEnumItemName functions allow you to convert the name of an enumerated value to an integer value, and to convert an integer value to the name of an enumerated value.

This example gets the numeric value for the boolean! enumerated value, then uses it to return the string value:

```pblong lType = session->GetEnumItemValue("object", boolean" ); // returns 138
LPCTSTR szEnum = session->GetEnumItemName( "object", lType ); // returns "boolean"
```

Notice that the second argument in the GetEnumItemValue call, the enumerated value, must not have an appended exclamation mark (!).

To return an enumerated value from an extension to PowerScript, use the SetLong function to set the value of the enumerated variable into IPB_Value (you cannot use SetInt or SetShort).

To obtain an enumerated variable’s value, you can use GetInt or GetShort as well as GetLong, as long as the value is in the appropriate range. For example, if you attempt to use GetInt to obtain a value that is more than 32767, the returned value is truncated.
CHAPTER 5
Calling PowerBuilder from C++

About this chapter
A third-party application or server written in C++ can load the
PowerBuilder VM, use PowerBuilder nonvisual objects, and use
PowerBuilder visual controls. This chapter uses some simple examples to
illustrate the process.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About calling PowerScript from C++ apps</td>
<td>65</td>
</tr>
<tr>
<td>Calling PowerBuilder objects from C++</td>
<td>66</td>
</tr>
<tr>
<td>Accessing result sets</td>
<td>72</td>
</tr>
<tr>
<td>Processing PowerBuilder messages in C++</td>
<td>73</td>
</tr>
<tr>
<td>More PNI possibilities</td>
<td>78</td>
</tr>
</tbody>
</table>

About calling PowerScript from C++ applications

If you have a PowerBuilder custom class user object that performs
intensive programming that would be useful to an application that you
need to write in C++, you can access the object directly from the C++
application using PNI. You do not need to make the user object into a
COM component or automation server.

To call functions on a PowerBuilder object, you can embed the PBVM in
the C++ application. The C++ application must load the PBVM by
loading pbvm125.dll with the Windows LoadLibrary function, get a pointer
to the IPB_VM interface by calling the PB_GetVM function exported by
pbvm125.dll, and create a session by calling the IPB_VM CreateSession
function.

The application can then create an instance of the PowerBuilder class and
invoke its functions through the IPB_Session interface.
Calling PowerBuilder objects from C++

The following figure illustrates the relationship between the C++ application and the interfaces provided by the PBVM.

Figure 5-1: Embedding the PBVM in a C++ application

Calling PowerBuilder objects from C++

This section presents a simple example that illustrates how to call a function on a PowerBuilder custom class user object from a C++ application:

- Creating a PowerBuilder object to be called from C++
- Getting the signature of a function
- Creating the C++ application
- Running the C++ application

Creating a PowerBuilder object to be called from C++

To keep the code for this example simple, create an application with one custom class user object that has one function. The function returns the product of two integers:

1. In PowerBuilder, create a new workspace.
2. Select the Application icon from the Target page of the New dialog box and name the application loadpbvm.
3. Select the Custom Class icon from the PB Object page of the New dialog box.
4 In the Function prototype window, create a function with this signature:

```c
f_mult ( integer arg1, integer arg2 )  returns integer
```

5 Save the user object as nvo_mult and close the User Object painter.

### Getting the signature of a function

To write the C++ code that invokes the `f_mult` function, you need to obtain its method ID. The method ID is used to initialize the PBCallInfo structure and to invoke the function. There are two IPB_Session functions that return a method ID: `GetMethodID`, which takes a signature, and `FindMatchingFunction`, which takes a comma-separated list of arguments. You use the same functions when you call PowerScript from the code in your extension; see “Calling PowerScript from an extension” on page 39.

If you want to use `GetMethodID`, you need a signature. This function is simple enough that you do not need a tool to obtain a signature—the signature is the string `III`, which indicates that the function returns an integer and takes two integers as arguments.

For more complicated functions, you can get the signature from the System Tree or with the `pbsig125` tool.

To get the signature of `f_mult` in the System Tree, expand `nvo_mult`, right-click on the `f_mult` function, and select Properties from the pop-up menu. The signature displays in the Properties dialog box in the Signature text box:
Calling PowerBuilder objects from C++

Getting a signature using pbsig125

To get the signature of f_mult with pbsig125, type the following at a command prompt:

```
pbsig125 d:\pbls\loadpbvm.pbl
```

In the output of pbsig125, the comment on the last line contains the signature to be passed as the method ID argument to GetMethodID:

```
PB Object Name: loadpbvm

PB Object Name: nvo_mult
  public function integer f_mult (integer arg1,
    integer arg2)
  /* III */
```

For more information about the pbsig125 tool and the format of method signatures, see pbsig125 on page 255.

Creating the C++ application

To create the C++ application, follow these steps:

1. Load the PowerBuilder VM
2. Call PB_GetVM to get a pointer to the IPB_VM interface
3. Create an IPB_Session object within IPB_VM
4. Create an instance of the PowerBuilder object
5. Initialize the PBCallInfo structure
6. Call the PowerBuilder function
7. Write cleanup code

Load the PowerBuilder VM

In your C++ development tool, create a new console application project. The `include` directory for the PBNI SDK, typically `PowerBuilder 12.5\SDK\PBNI\include`, must be in your include path. If you use any helper classes, the source file that contains them must be added to your project. For a list of files and helper classes, see the table in “The PBNI SDK” on page 8.
CHAPTER 5  Calling PowerBuilder from C++

The code for the C++ application creates an IPB_VM object using the PB_GetVM function and loads the PowerBuilder VM:

```c
#include "pbext.h"
#include "stdio.h"

typedef PBXEXPORT PBXRESULT (*P_PB_GetVM)(IPB_VM** vm);

int main(int argc, char *argv[])
{
    IPB_Session* session;
    IPB_VM* pbvm = NULL;

    // Load the PowerBuilder VM module
    HINSTANCE hinst = LoadLibrary("pbvm125.dll");
    if (hinst == NULL) return 0;
    fprintf(stderr, "Loaded PBVM successfully\n");

    Call PB_GetVM to get a pointer to the IPB_VM interface
    The next step is to call the PB_GetVM function to get a pointer to the IPB_VM interface:

    P_PB_GetVM getvm = (P_PB_GetVM)GetProcAddress(hinst,"PB_GetVM");
    if (getvm == NULL) return 0;
    getvm(&pbvm);
    if (pbvm == NULL) return 0;

    Create an IPB_Session object within IPB_VM
    Next create an IPB_Session object within IPB_VM, using the PowerBuilder application’s name and library list as arguments:

    // loadpbvm.pbl must contain an application object
    // named loadpbvm and it must be on the search path
    // for the executable file
    LPCTSTR LibList[] = {"loadpbvm.pbl"};
    if (pbvm->CreateSession("loadpbvm", LibList, 1,
                           &session) != PBX_OK )
    {
        fprintf(stderr, "Error in CreateSession\n");
        return 1;
    }

    fprintf(stderr, "Created session successfully\n");

```
Calling PowerBuilder objects from C++

Create an instance of the PowerBuilder object

After the session has been created, the C++ application can create PowerBuilder objects and call PowerBuilder functions in that session.

You use the FindGroup function to locate the group that contains the user object you want to use. FindGroup takes the name of the object as its first argument, and an enumerated type as its second argument. You are looking for a user object, so the second argument is pbgroup_userobject.

You pass the group returned from FindGroup to the FindClass function to get a class that you can pass to the NewObject function:

```cpp
// Create the PowerBuilder object contained in loadpbvm.pbl.
// First find the group that contains the user object nvo_mult
pbgroup group = session->FindGroup("nvo_mult", pbgroup_userobject);
if (group == NULL) return 0;

// Now find the class nvo_mult in the group
pbclass cls = session->FindClass(group,"nvo_mult");
if (cls == NULL) return 0;

// Create an instance of the PowerBuilder object
pbobject pbobj = session->NewObject(cls);
```

Initialize the PBCallInfo structure

Next, get the method ID for the function you want to call and initialize a PBCallInfo structure. You pass the signature obtained in “Getting the signature of a function” on page 67 to the GetMethodID function:

```cpp
// PBCallInfo contains arguments and return value
PBCallInfo ci;

// To call the class member function f_mult,
// pass its signature as the last argument
// to GetMethodID
pbmethodID mid = session->GetMethodID(cls, "f_mult", PBRT_FUNCTION, "III");

// Initialize call info structure based on method ID
session->InitCallInfo(cls, mid, &ci);
```
You could use `FindMatchingFunction` instead of `GetMethodID` to get the method ID. The call would look like this, because `f_mult` takes two integer arguments:

```cpp
pbmethodID mid = session->FindMatchingFunction(cls, "f_mult", PBRT_FUNCTION, "int, int");
```

**Call the PowerBuilder function**

Before you call the function, you must supply the integers to be multiplied. For the sake of simplicity, the following code sets them directly in the PBCallInfo structure.

```cpp
// Set IN arguments. The prototype of the function is
// integer f_mult(integer arg1, integer arg2)
ci.pArgs->GetAt(0)->SetInt(123);
ci.pArgs->GetAt(1)->SetInt(45);
```

Finally call the function, wrapping it in a try-catch statement to handle any runtime errors:

```cpp
// Call the function
try
{
    session->InvokeObjectFunction(pbobj, mid, &ci);

    // Was PB exception thrown?
    if (session->HasExceptionThrown())
    {
        // Handle PB exception
        session->ClearException();
    }
}
catch (...)
{
    // Handle C++ exception
}

// Get the return value and print it to the console
pbint ret = ci.returnValue->GetInt();
fprintf(stderr, "The product of 123 and 45 is %i\n", ret);
```
Accessing result sets

Write cleanup code

When you have finished with the PBCallInfo structure, call FreeCallInfo to release the memory allocated to it, then delete the structure, release the session, and free the library:

```cpp
// Release Call Info
session->FreeCallInfo(&ci);
delete &ci;

// Release session
session->Release();
return 0;
FreeLibrary(hinst);
}
```

Running the C++ application

When you run the compiled executable file at the command prompt, if the PowerBuilder VM is loaded and the session is created successfully, the following output displays in the command window:

```
Loaded PBVM successfully
Created session successfully
The product of 123 and 45 is 5535
```

Accessing result sets

You can use the IPB_ResultSetAccessor interface to access result sets in PowerBuilder. Use the IPB_Session GetResultSetAccessor method to create an instance of the interface using a result set returned from PowerBuilder as the method’s argument. You can then use the IPB_ResultSetAccessor’s getColumnCount, GetRowCount, GetItemData, and GetColumnMetaData methods to obtain information from the result set.

GetItemData uses the IPB_RSItemData interface to handle the data in each cell in the result set. If the data has a date, time, or datetime datatype, it is stored in a PB_DateData, PB_TimeData, or PB_DateTimeData structure.

To create a result set that can be passed to PowerBuilder, use the IPB_Session CreateResultSet method. See CreateResultSet on page 108 for an example.
Processing PowerBuilder messages in C++

You can open a PowerBuilder window from a C++ application or from an extension, but to make sure that events triggered in the window or control are processed, you need to make sure that the C++ application processes PowerBuilder messages. The IPB_Session ProcessPBMessage function lets you do this.

Each time the ProcessPBMessage function is called, it attempts to retrieve a message from the PowerBuilder message queue and process it. The function is similar to the PowerBuilder Yield function, which yields control to other graphic objects and pulls messages from PowerBuilder objects and other graphic objects from the queue. However, ProcessPBMessage processes only one message at a time, and it processes only PowerBuilder messages.

Messages are added to the PowerBuilder message queue when you call the PostEvent function.

You need to make sure that the ProcessPBMessage function is called repeatedly. For most C++ applications, you can provide a message loop in the main function and insert the IPB_Session ProcessPBMessage function in the message loop. This is shown in the example that follows.

If you use Microsoft Foundation Classes (MFC), you cannot modify the built-in message loop. To ensure that the ProcessPBMessage function is called repeatedly, you can overload the CWnd::WindowProc function and insert ProcessPBMessage into the overloaded function:

```cpp
LRESULT CCallPBVCtrl::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    d_session->ProcessPBMessage();
    return CDialog::WindowProc(message, wParam, lParam);
}
```

Examples

The following code fragments are from a C++ program that opens a window. The window has a menu item that invokes the Open event of a PowerBuilder application.
The call to `ProcessPBMessage` is in a loop in the `WinMain` function:

```cpp
int __stdcall WinMain(HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPSTR lpCmdLine,
                      int nCmdShow)
{
    MSG msg;

    WNDCLASSEX wcex;

    // initialization code omitted
    ...
    RegisterClassEx(&wcex);

    HWND hWnd = CreateWindow(szWndClsName,
                             "OpenPBWindow", WS_OVERLAPPEDWINDOW,
                             CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL,
                             hInstance, NULL);

    if (!hWnd)
    {
        return FALSE;
    }

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    try
    {
        while (GetMessage(&msg, NULL, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);

            // Call to ProcessPBMessage
            if (session)
                session->ProcessPBMessage();
        }
    }
    catch(...)
    {
        MessageBox(NULL, "Exception occurs",
                   "Exception", MB_OK);
    }
    return msg.wParam;
}
```
In the WndProc function, when the WM_CREATE message is passed, the PBVM is loaded and the library list, containing openwin.pbl, is passed to CreateSession. When the user selects the menu item that opens the PowerBuilder window, the FindGroup, FindClass, and GetMethodID functions obtain the information needed to create a new application object, initialize the PBCallInfo structure, and trigger the application object’s Open event:

```c
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
    case WM_CREATE:
        // Load the PBVM
        hPBVMInst = ::LoadLibrary("pbvm125.dll");
        P_PB_GetVM getvm = (P_PB_GetVM)
            GetProcAddress(hPBVMInst,"PB_GetVM");
        IPB_VM* vm = NULL;
        getvm(&vm);

        // Define the library list and create the session
        static const char *liblist[] = {"openwin.pbl"};
        vm->CreateSession("openwin", liblist, 1,
                &session);
        break;
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case ID_PB_VISUAL:
            {
                // Initialize PBCallInfo and trigger the
                // application open event
                try
                {
                    pbgroup group = session->FindGroup
                        (*openwin", pbgroup_application);
```
pbclass cls = session->FindClass(group, "openwin");
pbmethodID mid = session->GetMethodID
    (cls, "open", PBRT_EVENT, "QS");
pbobject obj = session->NewObject(cls);
PBCallInfo ci;
session->InitCallInfo(cls, mid, &ci);
session->TriggerEvent(obj, mid, &ci);
session->FreeCallInfo(&ci);
}  
catch(...)  
{  
    MessageBox(NULL, "Exception occurs", "Exception", MB_OK);  
}  
break;
}  
default:  
    return DefWindowProc(hWnd, message, wParam, lParam);
}  
break;
case WM_PAINT:  
    hdc = BeginPaint(hWnd, &ps);  
    RECT rt;  
    GetClientRect(hWnd, &rt);  
    EndPaint(hWnd, &ps);  
    break;
case WM_DESTROY:  
    session->Release();  
    session = NULL;  
    FreeLibrary(hPBVMInst);  
    PostQuitMessage(0);  
    break;
default:  
    return DefWindowProc(hWnd, message, wParam, lParam);
}  
return 0;  
}
Testing

You can test the ProcessPBMessage function with a simple PowerBuilder application like this one:

1. Create a PowerBuilder application called openwin in openwin.pbl.
2. Create a main window, w_main, with three buttons.
3. Insert a window-level function, of_setcolor, that takes three integers as arguments and has this script:
   
   ```
   this.backcolor = rgb(red, green, blue)
   ```
4. Insert a window-level user event, ue_test, with this script:
   
   ```
   MessageBox("ue_test", "This is a user event")
   ```
5. Provide the following scripts for the clicked events of the buttons:

   ```
   //cb_1:
   MessageBox("Button 1", "Clicked")
   parent.of_setcolor(255, 255, 0)

   //cb_2:
   MessageBox("Button 2", "Clicked")
   parent.PostEvent("ue_event") // not fired
   parent.of_setcolor(255, 0, 0)

   //cb_3:
   MessageBox("Button 3", "Clicked")
   cb_1.PostEvent(Clicked!) // not fired
   ```
6. Script the application’s Open event:

   ```
   open (w_main)
   ```

When the ProcessPBMessage function is included in the C++ application, the application runs from C++ as it does in PowerBuilder. The posted events in cb_2 and cb_3 are processed.

Now try commenting out these lines in the C++ application, recompiling, and running the application again:

```
if (session)
session->ProcessPBMessage();
```

The message boxes still display (response windows have their own message loop) and the of_setcolor function is called, but the posted events do not fire.
More PNI possibilities

The ability to create visual, nonvisual, and marshaler extensions, and to call PowerBuilder objects from external C++ applications, opens up numerous opportunities to combine these capabilities to develop more complex applications.

Writing an extension that loads the PBVM

Most of the examples in this book and on the PowerBuilder CodeXchange Web site at http://powerbuilder.codexchange.sybase.com show you how to create an extension in C++ and use it in PowerBuilder, or how to write a C++ application that loads the PowerBuilder VM.

You could also write an extension that loads the PowerBuilder VM and uses a custom class user object, using the techniques described in this chapter. The following figure depicts the interaction between the PBVM and an external application that uses an extension.

Figure 5-2: Interaction between PNI, the PBVM, and external applications

Calling PowerBuilder from Java

You can combine the ability to call PowerBuilder classes from C++, as described in this chapter, with the ability to create marshaler extensions, as described in Chapter 3, “Creating Marshaler Extensions,” to call PowerBuilder from Java.

One way to do this is to create a Java proxy class that declares static native methods that can be called to load the PBVM, create PowerBuilder sessions, create PowerBuilder objects, and invoke PowerScript functions. These native methods can call into the PBVM through PNI. Additional Java classes that represent the PBVM, PowerBuilder sessions, and PowerBuilder objects can be based on the proxy class.
The Java classes call the Java native methods through JNI, whereas the Java native methods call PowerBuilder through PNI.

There is a sample that illustrates these techniques on the PowerBuilder CodeXchange Web site at http://powerbuilder.codexchange.sybase.com.
More PBNI possibilities
This part contains reference information for PBNI datatypes, interfaces, and tools.
CHAPTER 6

PBN I Types and Return Values

About this chapter
This chapter contains information about the datatypes, enumerated types, and error return values used by the PowerBuilder Native Interface.

Contents
The following table maps PowerBuilder datatypes to predefined datatypes used in PBNI C++ modules.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerBuilder to PBNI datatype mappings</td>
<td>83</td>
</tr>
<tr>
<td>Types for access to PowerBuilder data</td>
<td>84</td>
</tr>
<tr>
<td>PBNI enumerated types</td>
<td>84</td>
</tr>
<tr>
<td>Error return values</td>
<td>86</td>
</tr>
</tbody>
</table>

PowerBuilder to PBNI datatype mappings

The following table maps PowerBuilder datatypes to predefined datatypes used in PBNI C++ modules.

<table>
<thead>
<tr>
<th>PowerBuilder datatype</th>
<th>Predefined datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>pbint</td>
</tr>
<tr>
<td>UInt</td>
<td>pbuint</td>
</tr>
<tr>
<td>Byte</td>
<td>pbbyte</td>
</tr>
<tr>
<td>Long</td>
<td>pblong</td>
</tr>
<tr>
<td>Longlong</td>
<td>pblonglong</td>
</tr>
<tr>
<td>Ulong</td>
<td>pbulong</td>
</tr>
<tr>
<td>Boolean</td>
<td>pbboolean</td>
</tr>
<tr>
<td>Real</td>
<td>pbreal</td>
</tr>
<tr>
<td>Double</td>
<td>pbdouble</td>
</tr>
<tr>
<td>Decimal</td>
<td>pbdec</td>
</tr>
<tr>
<td>Date</td>
<td>pbdate</td>
</tr>
<tr>
<td>Time</td>
<td>pbtime</td>
</tr>
<tr>
<td>DateTime</td>
<td>pbdatetime</td>
</tr>
<tr>
<td>Char</td>
<td>pbchar</td>
</tr>
<tr>
<td>Blob</td>
<td>pbblob</td>
</tr>
</tbody>
</table>
Types for access to PowerBuilder data

The types in the following table enable access to PowerBuilder data.

<table>
<thead>
<tr>
<th>PowerBuilder datatype</th>
<th>Predefined datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>pbstring</td>
</tr>
<tr>
<td>Powerobject</td>
<td>pboObject</td>
</tr>
</tbody>
</table>

### PBNI enumerated types

The `pbgroup_type` enumerated types are used in IPB_Session FindGroup calls to identify the type of group required.

<table>
<thead>
<tr>
<th>Value</th>
<th>PowerBuilder object</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbgroup_application</td>
<td>Application</td>
</tr>
<tr>
<td>pbgroup_datawindow</td>
<td>DataWindow definition</td>
</tr>
<tr>
<td>pbgroup_function</td>
<td>Global function</td>
</tr>
<tr>
<td>pbgroup_menu</td>
<td>Menu</td>
</tr>
<tr>
<td>pbgroup_proxy</td>
<td>Proxy definition for a remote object</td>
</tr>
<tr>
<td>pbgroup_structure</td>
<td>PowerBuilder structure type</td>
</tr>
<tr>
<td>pbgroup_userobject</td>
<td>PowerBuilder user object</td>
</tr>
<tr>
<td>pbgroup_window</td>
<td>Window</td>
</tr>
<tr>
<td>pbgroup_unknown</td>
<td>Unknown group</td>
</tr>
</tbody>
</table>
The `pbvalue_type` enumerated types are used in methods such as the IPB_Value
getValue method and the IPB_Session NewUnboundedSimpleArray method to
identify the type of PowerBuilder data.

**Table 6-4: Enumerated types for PowerBuilder values**

<table>
<thead>
<tr>
<th>Value</th>
<th>PowerBuilder datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbvalue_notype</td>
<td>Undetermined datatype.</td>
</tr>
<tr>
<td>pbvalue_int</td>
<td>Int</td>
</tr>
<tr>
<td>pbvalue_uint</td>
<td>Uint</td>
</tr>
<tr>
<td>pbvalue_byte</td>
<td>Byte</td>
</tr>
<tr>
<td>pbvalue_long</td>
<td>Long</td>
</tr>
<tr>
<td>pbvalue_longlong</td>
<td>Longlong</td>
</tr>
<tr>
<td>pbvalue_ulong</td>
<td>Ulong</td>
</tr>
<tr>
<td>pbvalue_real</td>
<td>Real</td>
</tr>
<tr>
<td>pbvalue_double</td>
<td>Double</td>
</tr>
<tr>
<td>pbvalue_dec</td>
<td>Decimal</td>
</tr>
<tr>
<td>pbvalue_string</td>
<td>String</td>
</tr>
<tr>
<td>pbvalue_boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>pbvalue_any</td>
<td>Any (changed to another datatype when set explicitly)</td>
</tr>
<tr>
<td>pbvalue_blob</td>
<td>Blob</td>
</tr>
<tr>
<td>pbvalue_date</td>
<td>Date</td>
</tr>
<tr>
<td>pbvalue_time</td>
<td>Time</td>
</tr>
<tr>
<td>pbvalue_datetime</td>
<td>DateTime</td>
</tr>
<tr>
<td>pbvalue_char</td>
<td>Char</td>
</tr>
</tbody>
</table>

The `pbrt_type` enumerated types are used in IPB_Session GetMethodID calls to
identify the type of routine required.

**Table 6-5: Enumerated types for PowerBuilder routines**

<table>
<thead>
<tr>
<th>Value</th>
<th>Routine type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBRT_FUNCTION</td>
<td>Function</td>
</tr>
<tr>
<td>PBRT_EVENT</td>
<td>Event</td>
</tr>
</tbody>
</table>
Error return values

The following table shows the PBXRESULT return values and error codes returned from PNI methods.

<table>
<thead>
<tr>
<th>Table 6-6: PBXResult return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of PBXResult</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>PBX_OK</td>
</tr>
<tr>
<td>PBX_SUCCESS</td>
</tr>
<tr>
<td>PBX_FAIL</td>
</tr>
<tr>
<td>PBX_E_NO_REGISTER_FUNCTION</td>
</tr>
<tr>
<td>PBX_E_REGISTRATION_FAILED</td>
</tr>
<tr>
<td>PBX_E_BUILD_GROUP_FAILED</td>
</tr>
<tr>
<td>PBX_E_INVALID_ARGUMENT</td>
</tr>
<tr>
<td>PBX_E_INVOKE_METHOD_INACCESSIBLE</td>
</tr>
<tr>
<td>PBX_E_INVOKE_Wrong_NUM_ARGS</td>
</tr>
<tr>
<td>PBX_E_INVOKE_REFARG_ERROR</td>
</tr>
<tr>
<td>PBX_E_INVOKE_METHOD_AMBIGUOUS</td>
</tr>
<tr>
<td>PBX_E_INVOKE_FAILURE</td>
</tr>
<tr>
<td>PBX_E_MISMATCHED_DATA_TYPE</td>
</tr>
<tr>
<td>PBX_E_OUTOF_MEMORY</td>
</tr>
<tr>
<td>PBX_E_GET_PBV_FAILED</td>
</tr>
<tr>
<td>PBX_E_NO_SUCH_CLASS</td>
</tr>
<tr>
<td>PBX_E_CAN_NOT_LOCATE_APPLICATION</td>
</tr>
<tr>
<td>PBX_E_INVALID_METHOD_ID</td>
</tr>
<tr>
<td>PBX_E_READONLY_ARGS</td>
</tr>
<tr>
<td>PBX_E_ARRAY_INDEX_OUTOF_BOUNDS</td>
</tr>
</tbody>
</table>
CHAPTER 7

PNI Interfaces, Structures, and Methods

About this chapter

This chapter contains reference information about the classes, structures, and methods of the PowerBuilder Native Interface.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header file contents</td>
<td>88</td>
</tr>
<tr>
<td>Class and interface summary</td>
<td>88</td>
</tr>
<tr>
<td>IPB_Arguments interface</td>
<td>90</td>
</tr>
<tr>
<td>IPB_ResultSetAccessor interface</td>
<td>92</td>
</tr>
<tr>
<td>IPB_RSItemData interface</td>
<td>95</td>
</tr>
<tr>
<td>IPB_Session interface</td>
<td>96</td>
</tr>
<tr>
<td>IPB_Value interface</td>
<td>202</td>
</tr>
<tr>
<td>IPB_VM interface</td>
<td>211</td>
</tr>
<tr>
<td>IPBX.Marshaler interface</td>
<td>215</td>
</tr>
<tr>
<td>IPBX_NonVisualObject interface</td>
<td>218</td>
</tr>
<tr>
<td>IPBX_UserObject interface</td>
<td>219</td>
</tr>
<tr>
<td>IPBX_VisualObject interface</td>
<td>221</td>
</tr>
<tr>
<td>PBArrayInfo structure</td>
<td>226</td>
</tr>
<tr>
<td>PBCallInfo structure</td>
<td>226</td>
</tr>
<tr>
<td>PB_DateData structure</td>
<td>227</td>
</tr>
<tr>
<td>PB_DateTimeData structure</td>
<td>227</td>
</tr>
<tr>
<td>PB_TimeData structure</td>
<td>227</td>
</tr>
<tr>
<td>PBX_DrawItemStruct structure</td>
<td>228</td>
</tr>
<tr>
<td>PBAArrayAccessor template class</td>
<td>229</td>
</tr>
<tr>
<td>PBBoundArrayCreator template class</td>
<td>232</td>
</tr>
<tr>
<td>PBBoundObjectArrayCreator class</td>
<td>235</td>
</tr>
<tr>
<td>PBOObjectArrayAccessor class</td>
<td>236</td>
</tr>
<tr>
<td>PBUnboundArrayCreator template class</td>
<td>238</td>
</tr>
<tr>
<td>PBUnboundObjectArrayCreator class</td>
<td>240</td>
</tr>
<tr>
<td>Exported methods</td>
<td>241</td>
</tr>
<tr>
<td>Method exported by PowerBuilder VM</td>
<td>252</td>
</tr>
</tbody>
</table>
### Header file contents

PNI classes and interfaces are defined in a set of header files.

- **pbni.h**: The classes, structures, and methods defined in the header file `pbni.h` allow PowerBuilder extension modules to interact with PowerBuilder. This file also includes the `pbarray.h`, `pbfield.h`, and `pbnimd.h` header files.

- **pbarray.h`, `pbfield.h`, `pbtraits.h`, and `pbnimd.h**`: `pbarray.h` contains helper classes that make it easier to create arrays and access data in them. `pbfield.h` contains a helper class that makes it easier to access fields. Both header files rely on `pbtraits.h`, which provides specializations for the Value enumerated types. `pbnimd.h` contains machine-specific datatype definitions. These files should not be included directly in your code.

- **pbext.h**: The classes, structures, and methods defined in the header file `pbext.h` must be implemented in PowerBuilder extension modules to allow PowerBuilder applications to use the extension modules. `pbext.h` includes `pbni.h` and `pbevtid.h`.

- **pbevtid.h**: `pbevtid.h` contains mappings from PowerBuilder event strings to internal event identifiers. These mappings allow the PBVM to automatically fire events that you include in the description of an extension. For more information, see “Event processing in visual extensions” on page 34.

- **pbrsa.h**: `pbrsa.h` contains structures and interfaces used to access data in DataStores and DataWindow controls.

### Class and interface summary

This table lists the classes and interfaces that make up PNI. After the table, the classes and interfaces are listed in alphabetical order. The methods for each class are listed in alphabetical order after the class description.

Several additional helper classes that are defined in `pbni.h` are not listed in the table. These helper classes include:

- PBArrInfoHolder and PBCallInfoHolder – used to hold a PBArrInfo or PBCallInfo variable and release it when it is out of scope
- PBEvtTrigger, PBObjFunctionInvoker, and PBGlobalFunctionInvoker – used to trigger events and call object and global functions
### Table 7-1: Pdni class and interface summary

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPB_Arguments interface</td>
<td>Used to access the arguments of the PBCallInfo structure.</td>
<td>pbni.h</td>
</tr>
<tr>
<td>IPB_ResultSetAccessor interface</td>
<td>Used to access data in a DataWindow or DataStore.</td>
<td>pbrsa.h</td>
</tr>
<tr>
<td>IPB_RSItemData interface</td>
<td>Used to set data values in a result set from a DataWindow or DataStore.</td>
<td>pbrsa.h</td>
</tr>
<tr>
<td>IPB_Session interface</td>
<td>Used to interoperate with PowerBuilder. An abstract interface, it defines</td>
<td>pbni.h</td>
</tr>
<tr>
<td></td>
<td>methods for accessing PowerScript data, calling PowerScript functions,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>catching and throwing PowerScript exceptions, and setting a marshaler to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>convert PowerBuilder data formats to the user’s communication protocol.</td>
<td></td>
</tr>
<tr>
<td>IPB_Value interface</td>
<td>Used to hold PowerBuilder data. IPB_Value contains information about each</td>
<td>pbni.h</td>
</tr>
<tr>
<td></td>
<td>variable, including its type, null flag, access privileges, array or simple</td>
<td></td>
</tr>
<tr>
<td></td>
<td>type, and reference type.</td>
<td></td>
</tr>
<tr>
<td>IPB_VM interface</td>
<td>Used to load PowerBuilder applications in third-party applications and</td>
<td>pbni.h</td>
</tr>
<tr>
<td></td>
<td>interoperate with the PowerBuilder virtual machine (PBVM).</td>
<td></td>
</tr>
<tr>
<td>PBArrayInfo structure</td>
<td>Used to hold information about arrays.</td>
<td>pbni.h</td>
</tr>
<tr>
<td>NBCallInfo structure</td>
<td>Used to hold arguments and return type information in function calls</td>
<td>pbni.h</td>
</tr>
<tr>
<td></td>
<td>between Pdni and PowerBuilder.</td>
<td></td>
</tr>
<tr>
<td>PB_DateData structure</td>
<td>Used to pass data of type Date in the SetData function in the</td>
<td>pbrsa.h</td>
</tr>
<tr>
<td></td>
<td>IPB_RSItemData interface.</td>
<td></td>
</tr>
<tr>
<td>PB_DateTimeData structure</td>
<td>Used to pass data of type DateTime in the SetData function in the</td>
<td>pbrsa.h</td>
</tr>
<tr>
<td></td>
<td>IPB_RSItemData interface.</td>
<td></td>
</tr>
<tr>
<td>PB_TimeData structure</td>
<td>Used to pass data of type Time in the SetData function in the</td>
<td>pbrsa.h</td>
</tr>
<tr>
<td></td>
<td>IPB_RSItemData interface.</td>
<td></td>
</tr>
<tr>
<td>PBX_DrawItemStruct structure</td>
<td>Used to hold the properties of an external visual control that you want</td>
<td>pbext.h</td>
</tr>
<tr>
<td></td>
<td>to draw using the PBX_DrawVisualObject function.</td>
<td></td>
</tr>
<tr>
<td>PBArrayAccessor template class</td>
<td>Used to access items in an array.</td>
<td>parray.h</td>
</tr>
<tr>
<td>PBOBJECTArrayAccessor class</td>
<td>Used to access items in an object array.</td>
<td>parray.h</td>
</tr>
<tr>
<td>PBBoundedArrayCreator template class</td>
<td>Used to create bounded arrays.</td>
<td>parray.h</td>
</tr>
<tr>
<td>PBBoundedObjectArrayCreator class</td>
<td>Used to create bounded object arrays.</td>
<td>parray.h</td>
</tr>
<tr>
<td>PBBoundedObjectArrayCreator class</td>
<td>Used to create unbounded arrays.</td>
<td>parray.h</td>
</tr>
<tr>
<td>PBUnboundedObjectArrayCreator class</td>
<td>Used to create unbounded object arrays.</td>
<td>parray.h</td>
</tr>
</tbody>
</table>
### IPB_Arguments interface

**IPB_Marshaler interface**

Used to invoke remote methods and convert PowerBuilder data formats to the user’s communication protocol. A marshaler extension is a PowerBuilder extension that acts as the bridge between PowerBuilder and other components, such as EJBs, Java classes, CORBA objects, Web services, and so on.

**IPB_NonVisualObject interface**

Inherits from IPB_UserObject and is the direct ancestor class of nonvisual PowerBuilder native classes.

**IPB_UserObject interface**

The ancestor class of PowerBuilder native classes. It has two functions, Destroy and Invoke.

**IPB_VisualObject interface**

Inherits from IPB_UserObject and is the direct ancestor class of visual PowerBuilder native classes.

**Exported methods**

Some exported methods must be implemented in PowerBuilder extension modules.

**Method exported by PowerBuilder VM**

The PB_GetVM method is exported by the PowerBuilder VM and is used to pass the IPB_VM interface to the user.

---

### IPB_Arguments interface

**Description**

The IPB_Arguments and IPB_Value interfaces are used to pass values between the PowerBuilder VM and PowerBuilder extension modules. Each argument is represented by a pointer to the IPB_Value interface.

**Methods**

The IPB_Arguments interface has two methods, GetAt and GetCount.

#### GetAt

**Description**

Returns a pointer to the IPB_Value interface representing an argument whose order in the list of arguments is indicated by a specified index.

**Syntax**

```
GetAt ( pbint index )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>A valid index into the PBCallInfo structure</td>
</tr>
</tbody>
</table>

**Return value**

IPB_Value*.

**Examples**

In the following code fragment, GetAt obtains the first value in the PBCallInfo structure. The value has been passed in from the calling function.
PBCallInfo ci;
LPCSTR myPBNIObj = NULL;
IPB_Value* pArg0 = ci->pArgs->GetAt(0);
if (!pArg0->IsNull())
{
    pbstring t = pArg0->GetString();
    if (t != NULL)
        myPBNIObj = session->GetString(t);
}

See also GetCount

GetCount

Description Obtains the number of arguments in an instance of PBCallInfo.
Syntax GetCount() 
Return value pbint.
Examples This example uses GetCount in a FOR loop used to process different argument types:

```c
int i;
for (i=0; i < ci->pArgs->GetCount();i++)
{
    pbuint ArgsType;
    if( ci->pArgs->GetAt(i) -> IsArray())
    
        pArguments[i].array_val =
        ci->pArgs->GetAt(i) -> GetArray();
        continue;
    
    if( ci->pArgs->GetAt(i) -> IsObject())
    {
        if (ci->pArgs->GetAt(i) -> IsNull())
            pArguments[i].obj_val=0;
        else
            pArguments[i].obj_val =
            ci->pArgs->GetAt(i) -> GetObject();
        continue;
    }
    ...
```

See also GetAt
IPB_ResultSetAccessor interface

Description
The IPB_ResultSetAccessor interface is used to access result sets in DataWindow and DataStore objects.

Methods
The IPB_ResultSetAccessor interface has six methods:
- AddRef
- GetColumnCount
- GetColumnMetaData
- GetItemData
- GetRowCount
- Release

AddRef
Description
When you call the CreateResultSet function of interface IPB_Session, you need to pass an argument of type IPB_ResultSetAccessor. The AddRef function is called on that argument and the Release function is called when the pbobject is destroyed.

Syntax
AddRef ( )

Return value
None.

See also
CreateResultSet
GetColumnCount

GetColumnCount
Description
Obtains the number of columns.

Syntax
GetColumnCount ( )

Return value
Unsigned long.

Examples
This statement stores the number of columns in *numCols:

```
*numCols = d_rsAccessor->GetColumnCount();
```

See also
CreateResultSet
GetRowCount
GetColumnMetaData

Description
Obtains a column's metadata. The column number of the first column is 1. Memory must be allocated for columnName before this function call. The pointer values can be null.

Syntax
GetColumnMetaData (unsigned long columnNum, LPTSTR columnName, pbvalue_type* type, unsigned long* width)

Argument | Description
--- | ---
columnNum | The number of the column for which you want to obtain metadata
columnName | The name of the specified column
type | A pointer to the type of the specified column
width | A pointer to the width of the specified column

Return value
None.

Examples
This example gets the number of columns in a result set and allocates an array to hold the types of each column:

```c
CRsltSet::CRsltSet(IPB_ResultSetAccessor* rsAccessor)
    :m_lRefCount (0), d_rsAccessor(rsAccessor)
{
    rsAccessor->AddRef();
    // for each column
    ULONG nNumColumns = d_rsAccessor->GetColumnCount();
    d_arrColTypes = new USHORT[nNumColumns + 1];
    for (ULONG nColumn=1; nColumn <= nNumColumns; ++nColumn)
    {
        // get the column type into the array
        pbvalue_type type;
        d_rsAccessor->GetColumnMetaData (nColumn, NULL, &type, NULL);
        d_arrColTypes[nColumn] = (USHORT)type;
    }
}
```

See also
CreateResultSet
GetColumnCount
GetItemData
GetRowCount
**IPB_ResultSetAccessor interface**

### GetItemData
Description
Accesses the data in a cell. The first row is 1 and the first column is 1.

**Syntax**
GetItemData(unsigned long *row, unsigned long *col, IPB_RSItemData *data)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*row</td>
<td>The row number of the cell</td>
</tr>
<tr>
<td>*col</td>
<td>The column number of the cell</td>
</tr>
<tr>
<td>*data</td>
<td>A pointer to an IPB_RSItemData structure</td>
</tr>
</tbody>
</table>

**Return value**
Boolean.

**Examples**
This example stores the data in the first row and column in the IPB_RSItemData structure *sd*:
```c
    d_rsAccessor->GetItemData(1, 1, &sd);
```

**Usage**
If the value of *data* is null, this function issues the callback *data-*`->SetNull`. If the value is not null, it issues the callback *data-*`->SetData`. For more information, examine the IPB_RSItemData interface.

**See also**
CreateResultSet
GetColumnCount
GetColumnMetaData
GetRowCount
IPB_RSItemData interface
SetData
SetNull

### GetRowCount
**Description**
Obtains the number of rows.

**Syntax**
GetRowCount ()

**Return value**
Unsigned long.

**Examples**
This statement stores the number of rows in *numRows*:
```c
    *numRows = d_rsAccessor->GetRowCount();
```

**See also**
CreateResultSet
GetColumnCount
GetColumnMetaData
GetItemData
Release
Description When you call the CreateResultSet function of interface IPB_Session, you need to pass an argument of type IPB_ResultSetAccessor. The AddRef function is called on that argument and the Release function is called when the pbobject is destroyed.
Syntax Release ( )
Return value None.
See also AddRef CreateResultSet

IPB_RSItemData interface
Description The IPB_RSItemData interface is used as an argument to the GetItemData function of IPB_ResultSetAccessor.
Methods The IPB_RSItemData interface has two methods: SetData and SetNull.

SetData
Description Sets the data in an IPB_RSItemData structure when the GetItemData function of IPB_ResultSetAccessor is called and the data value is not null.
Syntax SetData(unsigned long len, void* data)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>The length of the data</td>
</tr>
<tr>
<td>data</td>
<td>A void pointer to the address of the data</td>
</tr>
</tbody>
</table>

Return value None.
Usage If the cell datatype is:
- string and decimal, the address points to a string
- date, the address points to a PB_DateData structure
- time, the address points to a PB_TimeData structure
- datetime, the address points to a PB_DateTimeData structure
- another datatype, the address points to data of the corresponding type
**SetNull**

Description: Sets the data in an IPB_RSItemData structure to null when the GetItemData function of IPB_ResultSetAccessor is called and the data value is not null.

Syntax: `SetNull()`

Return value: None.

See also: GetItemData, SetData

---

**IPB_Session interface**

Description: The IPB_Session interface is used to interoperate with PowerBuilder. An abstract interface, it defines methods for accessing PowerScript data, calling PowerScript functions, catching and throwing PowerScript exceptions, and setting a marshaler to convert PowerBuilder data formats to the user’s communication protocol.

Methods: This table lists functions by category. Full descriptions in alphabetic order follow the table.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Managing sessions</td>
<td>Release</td>
<td>Releases this IPB session. The IPB_Session object becomes invalid after the call.</td>
</tr>
<tr>
<td>Managing object references</td>
<td>AddGlobalRef</td>
<td>Adds a global reference to the specified PowerBuilder object.</td>
</tr>
<tr>
<td></td>
<td>AddLocalRef</td>
<td>Adds a local reference to the specified PowerBuilder object.</td>
</tr>
<tr>
<td></td>
<td>NewObject</td>
<td>Creates a new object of the specified type.</td>
</tr>
<tr>
<td></td>
<td>PopLocalFrame</td>
<td>Pops the current local reference frame from the current native method stack frame.</td>
</tr>
</tbody>
</table>
### Purpose

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PushLocalFrame</td>
<td>Pushes a local reference frame onto the current native method stack frame.</td>
</tr>
<tr>
<td>RemoveGlobalRef</td>
<td>Removes a global reference to the specified PowerBuilder object.</td>
</tr>
<tr>
<td>RemoveLocalRef</td>
<td>Removes a local reference to the specified PowerBuilder object.</td>
</tr>
</tbody>
</table>

### Managing shared properties

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetProp</td>
<td>Retrieves a pointer to the data value of a variable that has been registered as a shared property for the current IPB session.</td>
</tr>
<tr>
<td>RemoveProp</td>
<td>Removes the specified variable from the list of properties of the current IPB session.</td>
</tr>
<tr>
<td>SetProp</td>
<td>Adds a new variable to the list of properties of the current session or changes the value of an existing variable.</td>
</tr>
</tbody>
</table>

### Handling the PowerBuilder message queue

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcessPBMessage</td>
<td>Checks the PowerBuilder message queue and, if there is a message in the queue, attempts to process it.</td>
</tr>
</tbody>
</table>

### Handling exceptions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClearException</td>
<td>Clears the current PowerBuilder exception object.</td>
</tr>
<tr>
<td>GetException</td>
<td>Obtains the current thrown exception object.</td>
</tr>
<tr>
<td>HasExceptionThrown</td>
<td>Checks for the existence of an exception that has been thrown but not cleared.</td>
</tr>
<tr>
<td>ThrowException</td>
<td>Throws a PowerBuilder exception or inherited exception, replacing the existing exception if one exists.</td>
</tr>
</tbody>
</table>

### Passing arguments

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add&lt;type&gt;Argument</td>
<td>Adds an argument in a variable argument PowerBuilder call.</td>
</tr>
<tr>
<td>FreeCallInfo</td>
<td>Frees memory allocated by InitCallInfo.</td>
</tr>
<tr>
<td>InitCallInfo</td>
<td>Initializes the PBCallInfo structure.</td>
</tr>
</tbody>
</table>

### Finding PowerBuilder classes and objects

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FindGroup</td>
<td>Searches for a group with a given name and group type in the current library list.</td>
</tr>
<tr>
<td>FindClass</td>
<td>Searches for a class with a given name within a given group.</td>
</tr>
<tr>
<td>FindClassByClassID</td>
<td>Searches for a class with a given name and a given ID.</td>
</tr>
<tr>
<td>GetClass</td>
<td>Returns the class handle of a PowerBuilder object.</td>
</tr>
<tr>
<td>GetClassName</td>
<td>Returns the name of a class in lowercase.</td>
</tr>
<tr>
<td>GetCurrGroup</td>
<td>Returns the name of the current group.</td>
</tr>
<tr>
<td>GetSuperClass</td>
<td>Returns the base class of a class, if any.</td>
</tr>
</tbody>
</table>
### IPB_Session interface

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GetSystemClass</td>
<td>Returns the system class handle of a PowerBuilder object.</td>
</tr>
<tr>
<td></td>
<td>GetSystemGroup</td>
<td>Returns the class that contains all the system global functions.</td>
</tr>
<tr>
<td></td>
<td>IsAutoInstantiate</td>
<td>Returns true if the specified class is an autoinitialized class; otherwise returns false.</td>
</tr>
<tr>
<td>Working with functions and events</td>
<td>FindMatchingFunction</td>
<td>Finds a function that has the specified argument list.</td>
</tr>
<tr>
<td></td>
<td>GetMethodID</td>
<td>Returns the ID of the requested function.</td>
</tr>
<tr>
<td></td>
<td>GetMethodIDByEventID</td>
<td>Returns the ID of the function that has a given predefined PowerBuilder event ID.</td>
</tr>
<tr>
<td></td>
<td>InvokeClassFunction</td>
<td>Invokes system or user global functions.</td>
</tr>
<tr>
<td></td>
<td>InvokeObjectFunction</td>
<td>Invokes a class member function.</td>
</tr>
<tr>
<td></td>
<td>TriggerEvent</td>
<td>Triggers a PowerBuilder event.</td>
</tr>
<tr>
<td>Working with enumerated variables</td>
<td>GetEnumItemName</td>
<td>Obtains the name of an enumerated variable.</td>
</tr>
<tr>
<td></td>
<td>GetEnumItemValue</td>
<td>Obtains the value of an enumerated variable.</td>
</tr>
<tr>
<td>Working with global variables</td>
<td>GetGlobalVarID</td>
<td>Returns the name of a global variable.</td>
</tr>
<tr>
<td></td>
<td>GetGlobalVarType</td>
<td>Returns the datatype of a global variable.</td>
</tr>
<tr>
<td></td>
<td>Get&lt;type&gt;GlobalVar</td>
<td>Returns the value of a global variable of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>GetPBAAnyGlobalVar</td>
<td>Obtains the value of a global variable of type Any.</td>
</tr>
<tr>
<td></td>
<td>IsGlobalVarArray</td>
<td>Returns true if the global variable contains an array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsGlobalVarNull</td>
<td>Returns true if the global variable contains a null value, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsGlobalVarObject</td>
<td>Returns true if the global variable contains a pbobject, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>Set&lt;type&gt;GlobalVar</td>
<td>Sets the value of a global variable of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>SetGlobalVarToNull</td>
<td>Sets the value of a shared variable to null.</td>
</tr>
<tr>
<td>Working with shared variables</td>
<td>GetSharedVarID</td>
<td>Returns the name of a shared variable.</td>
</tr>
<tr>
<td></td>
<td>GetSharedVarType</td>
<td>Returns the datatype of a shared variable.</td>
</tr>
<tr>
<td></td>
<td>Get&lt;type&gt;SharedVar</td>
<td>Returns the value of a shared variable of a specific datatype.</td>
</tr>
</tbody>
</table>
### Purpose Method Description

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GetPBAryAnySharedVar</td>
<td>Obtains the value of a shared variable of type Any.</td>
</tr>
<tr>
<td></td>
<td>IsSharedVarArray</td>
<td>Returns true if the shared variable contains an array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsSharedVarNull</td>
<td>Returns true if the shared variable contains a null value, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsSharedVarObject</td>
<td>Returns true if the shared variable contains a pbobject, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>Set&lt;type&gt;SharedVar</td>
<td>Sets the value of a shared variable of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>SetSharedVarToNull</td>
<td>Sets the value of a shared variable to null.</td>
</tr>
<tr>
<td>Working with arrays</td>
<td>Get&lt;type&gt;ArrayItem</td>
<td>Returns the value of an array item of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>GetArrayInfo</td>
<td>Obtains information about an array.</td>
</tr>
<tr>
<td></td>
<td>GetArrayItemType</td>
<td>Obtains the datatype of an item in an array.</td>
</tr>
<tr>
<td></td>
<td>GetArrayLength</td>
<td>Returns the length of an array.</td>
</tr>
<tr>
<td></td>
<td>GetPBAryAnyArrayItem</td>
<td>Obtains the value of an array item of type Any.</td>
</tr>
<tr>
<td></td>
<td>IsArrayItemNull</td>
<td>Returns true if the array item contains an array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>NewBoundedSimpleArray</td>
<td>Creates a bounded simple data array.</td>
</tr>
<tr>
<td></td>
<td>NewUnboundedSimpleArray</td>
<td>Creates an unbounded simple data array.</td>
</tr>
<tr>
<td></td>
<td>NewBoundedObjectArray</td>
<td>Creates a bounded PowerBuilder object or structure array.</td>
</tr>
<tr>
<td></td>
<td>NewUnboundedObjectArray</td>
<td>Creates an unbounded PowerBuilder object or structure data array.</td>
</tr>
<tr>
<td></td>
<td>ReleaseArrayInfo</td>
<td>Releases memory returned by GetArrayInfo.</td>
</tr>
<tr>
<td></td>
<td>Set&lt;type&gt;ArrayItem</td>
<td>Sets the value of an array item of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>SetArrayItemToNull</td>
<td>Sets the value of an array item to null.</td>
</tr>
<tr>
<td>Working with strings</td>
<td>GetStringLength</td>
<td>Returns the length of a string in bytes without the terminator.</td>
</tr>
<tr>
<td></td>
<td>GetString</td>
<td>Returns a pointer to the string passed in as an argument.</td>
</tr>
<tr>
<td></td>
<td>NewString</td>
<td>Creates a new string.</td>
</tr>
<tr>
<td></td>
<td>ReleaseString</td>
<td>Releases the memory used by a string.</td>
</tr>
<tr>
<td></td>
<td>SetString</td>
<td>Frees an existing string and assigns a new string value to it.</td>
</tr>
</tbody>
</table>
## IPB_Session interface

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working with binary large</td>
<td>GetBlob</td>
<td>Returns a pointer to the data buffer for a blob.</td>
</tr>
<tr>
<td>objects</td>
<td>GetBlobLength</td>
<td>Returns the length in bytes of blob data in a buffer.</td>
</tr>
<tr>
<td></td>
<td>NewBlob</td>
<td>Creates a new blob and duplicates a buffer for the new blob data.</td>
</tr>
<tr>
<td></td>
<td>SetBlob</td>
<td>Destroys the existing data in a blob and copies data into it from a buffer.</td>
</tr>
<tr>
<td>Working with decimal</td>
<td>GetDecimalString</td>
<td>Converts decimal data in a pbdec object to a string.</td>
</tr>
<tr>
<td>values</td>
<td>NewDecimal</td>
<td>Allocates resources for a new decimal data object.</td>
</tr>
<tr>
<td></td>
<td>ReleaseDecimalString</td>
<td>Frees the memory acquired using GetDecimalString.</td>
</tr>
<tr>
<td></td>
<td>SetDecimal</td>
<td>Converts a string to a decimal.</td>
</tr>
<tr>
<td>Working with date and time</td>
<td>GetDateString</td>
<td>Converts data in a pbdate object to a string.</td>
</tr>
<tr>
<td>values</td>
<td>GetDateTimeString</td>
<td>Converts data in a pbdatetime object to a string.</td>
</tr>
<tr>
<td></td>
<td>NewDate</td>
<td>Creates a new pbdate data object.</td>
</tr>
<tr>
<td></td>
<td>NewDateTime</td>
<td>Creates a new pbdatetime data object.</td>
</tr>
<tr>
<td></td>
<td>NewTime</td>
<td>Creates a new pbtime data object.</td>
</tr>
<tr>
<td></td>
<td>ReleaseDateString</td>
<td>Frees the memory acquired using GetDateString.</td>
</tr>
<tr>
<td></td>
<td>ReleaseDateTimeString</td>
<td>Frees the memory acquired using GetDateTimeString.</td>
</tr>
<tr>
<td></td>
<td>ReleaseTimeString</td>
<td>Frees the memory acquired using GetTimeString.</td>
</tr>
<tr>
<td></td>
<td>SetDate</td>
<td>Resets the value of the specified pbdate object.</td>
</tr>
<tr>
<td></td>
<td>SetDateTime</td>
<td>Resets the value of the specified pbdatetime object.</td>
</tr>
<tr>
<td></td>
<td>SetTime</td>
<td>Resets the value of the specified pbtime object.</td>
</tr>
<tr>
<td></td>
<td>SplitDate</td>
<td>Splits the specified pbdate object into a year, month, and day.</td>
</tr>
<tr>
<td></td>
<td>SplitDateTime</td>
<td>Splits the specified pbdatetime object into a year, month, and day.</td>
</tr>
<tr>
<td></td>
<td>SplitTime</td>
<td>Splits the specified pbtime object into a year, month, and day.</td>
</tr>
<tr>
<td>Purpose</td>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Working with data values</td>
<td>AcquireArrayItemValue</td>
<td>Clones the data in the PBCallInfo structure in an array item and resets the IPB_Value pointer.</td>
</tr>
<tr>
<td></td>
<td>AcquireValue</td>
<td>Clones the data in the PBCallInfo structure and resets the IPB_Value pointer.</td>
</tr>
<tr>
<td></td>
<td>ReleaseValue</td>
<td>Frees the value acquired by the AcquireValue or AcquireArrayItemValue method.</td>
</tr>
<tr>
<td></td>
<td>SetValue</td>
<td>Sets the value of one IPB_Value object to the value of another IPB_Value object</td>
</tr>
<tr>
<td>Working with fields</td>
<td>GetFieldID</td>
<td>Obtains the internal ID of a class instance variable.</td>
</tr>
<tr>
<td></td>
<td>GetFieldName</td>
<td>Obtains the name of the specified field.</td>
</tr>
<tr>
<td></td>
<td>GetFieldType</td>
<td>Obtains the datatype of a class instance variable.</td>
</tr>
<tr>
<td></td>
<td>GetNumOfFields</td>
<td>Obtains the number of fields in the specified class.</td>
</tr>
<tr>
<td></td>
<td>GetPBA nyField</td>
<td>Obtains the value of a variable of type Any.</td>
</tr>
<tr>
<td></td>
<td>Get&lt;type&gt;Field</td>
<td>Obtains a pointer to the instance variable data for a specified variable.</td>
</tr>
<tr>
<td></td>
<td>IsFieldArray</td>
<td>Returns true if the field contains an array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsFieldNull</td>
<td>Returns true if the field contains a null value array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsFieldObject</td>
<td>Returns true if the field contains a pbobject, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>Set&lt;type&gt;Field</td>
<td>A set of datatype-specific functions. Sets the value of an instance field of an object.</td>
</tr>
<tr>
<td></td>
<td>UpdateField</td>
<td>Refreshes a visual property of a PowerBuilder object.</td>
</tr>
<tr>
<td>Working with native classes</td>
<td>GetNativeInterface</td>
<td>Obtains a pointer to the interface of a native class.</td>
</tr>
<tr>
<td></td>
<td>IsNativeObject</td>
<td>Determines whether a pbobject is an instance of a native class.</td>
</tr>
<tr>
<td>Accessing result sets from DataWindows and DataStores</td>
<td>CreateResultSet</td>
<td>Creates a result set object using a pointer to an IPB_ResultSetAccessor object.</td>
</tr>
<tr>
<td></td>
<td>GetResultSetAccessor</td>
<td>Obtains an interface through which you can read data from a result set.</td>
</tr>
<tr>
<td></td>
<td>ReleaseResultSetAccessor</td>
<td>Releases the pointer obtained using GetResultSetAccessor.</td>
</tr>
</tbody>
</table>
IPB_Session interface

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working with marshaler</td>
<td>GetMarshaler</td>
<td>Obtains the marshaler object associated with a proxy object.</td>
</tr>
<tr>
<td>extensions</td>
<td>NewProxyObject</td>
<td>Creates a proxy for a remote object.</td>
</tr>
<tr>
<td></td>
<td>SetMarshaler</td>
<td>Sets a marshaler that will be used to invoke remote methods and convert PowerBuilder data formats to the user’s communication protocol.</td>
</tr>
</tbody>
</table>

**AcquireArrayItemValue**

**Description**
Clones the data in the PBCallInfo structure in an array item and resets the IPB_Value pointer.

**Syntax**

```c
AcquireArrayItemValue( pbarray array, pblong dim[ ])  
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid pbarray structure.</td>
</tr>
<tr>
<td>dim</td>
<td>A pblong array to hold the indexes of all dimensions of the array. The size of the array must equal the dimensions of array.</td>
</tr>
</tbody>
</table>

**Return value**
IPB_Value*.

**Examples**
This FOR loop acquires the value of an item in an array and sets the value in another array:

```c
for( i=1; i <= bound; i++)  
{  
    dim[0]= i;  
    ipv = Session -> AcquireArrayItemValue(refArg, dim);  
    Session -> SetArrayItemValue(*i_array, dim, ipv);  
    Session -> ReleaseValue(ipv);  
}
```

**Usage**
The AcquireArrayItemValue method enables you to retain the data in the PBCallInfo structure for a single array item.

The AcquireArrayItemValue method is independent of the type of the data but is most useful for acquiring the value of pointer values, such as pbvalue_string, pbvalue_blob, and so on. When you call FreeInfo, the data is not freed and the pointer returned by AcquireArrayItemValue is still valid.

When you no longer need the data, you must call the ReleaseValue method to free the data. Failing to do so causes a memory leak.
The PBVM clones a new IPB_Value and resets the existing one. If you attempt to get or acquire the original value, the value returned is zero or null until another IPB_Value is set to the value.

**Working with large arrays**
The processing that the AcquireArrayItemValue and ReleaseValue methods perform results in poor performance when handling large arrays. It is more efficient to get the type of the array and handle each type with appropriate type-specific functions.

See also ReleaseValue

### AcquireValue
**Description**
Clones the data in the PBCallInfo structure and resets the IPB_Value pointer.

**Syntax**
```
AcquireValue ( IPBValue* value )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The value to be returned</td>
</tr>
</tbody>
</table>

**Return value**
IPB_Value*.

**Examples**
The AcquireValue method is used to obtain a message argument value. Later, when the value is no longer needed, it is released using ReleaseValue to avoid memory leaks:

```
// Acquire a value
MessageArg = session->AcquireValue
   ( ci->pArgs->GetAt(0) );
pbstring pbMessage = MessageArg->GetString() ;
Message = (LPSTR)session->GetString(pbMessage) ;

// Cleanup phase
if (MessageArg)
{
   Session->ReleaseValue ( MessageArg ) ;
}
```

**Usage**
The AcquireValue method enables you to retain the data in the PBCallInfo structure. The AcquireValue method is independent of the type of the data but is most useful for acquiring the value of pointer values such as pbvalue_string, pbvalue_blob, and so on. When you call FreeInfo, the data is not freed and the pointer returned by AcquireValue is still valid.
If the value acquired is an array, the entire array is acquired. To acquire a single element in an array, use the AcquireItemValue method.

When you no longer need the data, you must call the ReleaseValue method to free the data. Failing to do so causes a memory leak.

The PBVM clones a new IPB_Value and resets the existing one. If you attempt to get or acquire the original value, the value returned is zero or null until another IPB_Value is set to the value.

See also
- AcquireArrayItemValue
- ReleaseValue

### Add<type>Argument

**Description**

Adds an argument of a specific type in a variable argument PowerBuilder call.

**Syntax**

AddArrayArgument (PBCallInfo *ci, pbblob value, pbboolean IsNull)
AddBlobArgument (PBCallInfo *ci, pbblob value, pbboolean IsNull)
AddBoolArgument (PBCallInfo *ci, pbboolean value, pbboolean IsNull)
AddByteArgument (PBCallInfo *ci, pbbyte value, pbboolean IsNull)
AddCharArgument (PBCallInfo *ci, pbchar value, pbboolean IsNull)
AddDateArgument (PBCallInfo *ci, pbdate value, pbboolean IsNull)
AddDateTimeArgument (PBCallInfo *ci, pbdatetime value, pbboolean IsNull)
AddDecArgument (PBCallInfo *ci, pbdec value, pbboolean IsNull)
AddDoubleArgument (PBCallInfo *ci, pbdouble value, pbboolean IsNull)
AddIntArgument (PBCallInfo *ci, pbint value, pbboolean IsNull)
AddLongArgument (PBCallInfo *ci, pblong value, pbboolean IsNull)
AddLongLongArgument (PBCallInfo *ci, pblonglong value, pbboolean IsNull)
AddObjectArgument (PBCallInfo *ci, pbobject value, pbboolean IsNull)
AddPBStringArgument (PBCallInfo *ci, pbstring value, pbboolean IsNull)
AddRealArgument (PBCallInfo *ci, pbreal value, pbboolean IsNull)
AddStringArgument (PBCallInfo *ci, LPCTSTR value, pbboolean IsNull)
AddTimeArgument (PBCallInfo *ci, pbttime value, pbboolean IsNull)
AddUintArgument (PBCallInfo *ci, pbuint value, pbboolean IsNull)
AddUlongArgument (PBCallInfo *ci, pbulong value, pbboolean IsNull)
PBXRESULT. PBX_OK on success.

Examples
This code tests that adding an integer argument to a PBCallInfo structure ci works correctly:

```c
long Cmy_pbni::f_Retrieve(IPB_Session* session, pbint retrieve_args, pbobject dwobj)
{
    pbclass cls;
    pbmethodID mid;
    PBCallInfo* ci = new PBCallInfo;
    pblong ret_val;
    PBXRESULT ret;

    cls = session->GetClass(dwobj);
    mid = session->GetMethodID
        (cls, "retrieve", PBRT_FUNCTION, "LAV");
    if (mid == kUndefinedMethodID)
        return -1;

    session->InitCallInfo(cls, mid, ci);
    ci->pArgs->GetAt(0)->SetInt(retrieve_args);
    session->AddIntArgument(ci, retrieve_args, false);
    ret = session->InvokeObjectFunction(dwobj, mid, ci);
    if (ret != PBX_OK)
        ret_val = ret;
    else
        ret_val = ci->returnValue->GetLong();

    session->FreeCallInfo(ci);
    delete ci;

    return ret_val;
}
```
IPB_Session interface

Usage
This call is used in variable argument PowerBuilder calls, such as
datawindow.retrieve(arg). After the call, the value returned by ci->pArgs->GetCount() increases by one.

See also
GetCount
InvokeClassFunction
InvokeObjectFunction

AddGlobalRef

Description
Adds a global reference to the specified PowerBuilder object.

Syntax
AddGlobalRef (pbobject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid PowerBuilder object handle</td>
</tr>
</tbody>
</table>

Return value
pbclass or null on error.

Examples
This example checks whether a return value is null, and if it is not, adds a global reference to it to the session:

    if (ci-> returnValue-> IsNull())
        ret_val = 0;
    else
    {
        ret_val = ci-> returnValue-> GetObject();
        Session -> AddGlobalRef(ret_val);
    }

See also
RemoveGlobalRef
AddLocalRef

Description
Adds a local reference to the specified PowerBuilder object.

Syntax
AddLocalRef (pbobject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid PowerBuilder object handle</td>
</tr>
</tbody>
</table>

Return value
pbclass or null on error.

Examples
This example defines functions that add and remove local references:

```c
void MyPBNIClass::reference()
{
    d_session->AddLocalRef(d_pbobject);
}

void MyPBNIClass::unreference()
{
    if(d_pbobject != NULL)
        d_session->RemoveLocalRef(d_pbobject);
}
```

See also
PopLocalFrame
PushLocalFrame
RemoveLocalRef

ClearException

Description
Clears the current PowerBuilder exception object.

Syntax
ClearException ()

Return value
None.

Usage
HasExceptionThrown returns false after a call to ClearException. If no exception has been thrown, this call has no effect.

See also
GetException
HasExceptionThrown
ThrowException
CreateResultSet

Description
Creates a result set object using a pointer to an IPB_ResultSetAccessor object.

Syntax
CreateResultSet (IPB_ResultSetAccessor* rs)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>A pointer to an IPB_ResultSetAccessor object</td>
</tr>
</tbody>
</table>

Return value
pobject.

Examples
This example loads the PBVM and calls the f_ret and f_in functions in the custom class user object n_rs in the PBL pbrs.pbl. The PowerScript for the functions is shown after the C++ code:

```cpp
#include "stdafx.h"
#include "windows.h"
#include "pbni.h"
#include "vector"
using std::vector;

void main(int argc, char* argv[]) {
    HINSTANCE hinst = LoadLibrary("pbvm125.dll");
    typedef PBXRESULT (*P_PB_GetVM) (IPB_VM** vm);
    P_PB_GetVM getvm = (P_PB_GetVM)GetProcAddress(hinst, "PB_GetVM");
    IPB_VM* pbvm;
    getvm(&pbvm);
    IPB_Session* session = NULL;
    vector<LPCSTR> ll(1);
    ll[0] = "pbrs.pbl";
    pbvm->CreateSession("pbrs", &ll[0], 1, &session);
    pbgroup group = session->FindGroup("n_rs", pbgroup_userobject);
    if (group == NULL) return;
    pbclass cls = session->FindClass(group, "n_rs");
    if (cls == NULL) return;
    pobject obj = session->NewObject(cls);
```
if (obj == NULL) return;

pbmethodID mid = session->GetMethodID(cls, "f_ret", 
PBR.Function, "Cresultset.");
PBCallInfo ci;
session->InitCallInfo(cls, mid, &ci);
session->InvokeObjectFunction(obj, mid, &ci);

// Use the result set returned from f_ret to
// create an IPB_ResultSetAccessor rsa
pbobject rs = ci.returnValue->GetObject();
IPB_ResultSetAccessor* rsa = 
    session->GetResultSetAccessor(rs);

// Create a result set object from rsa
pbobject rsobj = session->CreateResultSet(rsa);

// Call the f_in method
mid = session->GetMethodID(cls, "f_in", 
PBR.Function, "IRCresultset.");
PBCallInfo ci1;
session->InitCallInfo(cls, mid, &ci1);
// Set the result set object rsobj as the
// argument for f_in
    ci1.pArgs->GetAt(0)->SetObject(rsobj);
    session->InvokeObjectFunction(obj, mid, &ci1);

    session->FreeCallInfo(&ci);
    session->FreeCallInfo(&ci1);
}

f_ret retrieves data from a database into a DataStore and generates a result set:

ResultSet rs
DataStore ds

Long sts
Integer li_ret

// Profile EAS Demo DB V125
SQLCA.DBMS = "ODBC"
SQLCA.AutoCommit = False
SQLCA.DBParm = &
    "ConnectString='DSN=EAS Demo DB V125;UID=dba;PWD=sql'"
connect using sqlca;

ds = Create DataStore
ds.DataObject = ""
ds.DataObject = "d_rs"
ds.SetTransObject(sqlca)
w_main.dw_1.SetTransObject(sqlca)

long ll_ret, rows, rows2
ll_ret = ds.Retrieve()
ll_ret = w_main.dw_1.Retrieve()
//ds.shareddata(w_main.dw_1)
rows = ds.RowCount()
rows2 = w_main.dw_1.RowCount()
messagebox("info from f_ret", " row count is " & 
+ string(rows) + " or " + string(rows2))
sts = ds.GenerateResultSet(rs)

Return rs

f_in takes a result set, rs, as an argument and uses it to create a DataStore:

DataStore ds
Int cnt, li_ret

ds = Create DataStore
ds.CreateFrom(rs)
cnt = ds.RowCount()
messagebox("info from f_in", "row count is " +
string(cnt))
Return cnt

Usage To use the IPB_ResultSetAccessor interface, load the PBVM, obtain a result set from a PowerBuilder application, and call GetResultSetAccessor on this result set to get an IPB_ResultSetAccessor interface object. You can then call the methods of this object to get information about the result set. You can also call CreateResultSet using this object as an argument to create a result set that you can return to PowerBuilder.

When you call CreateResultSet, the AddRef function of the IPB_ResultSetAccessor interface is called on the rs argument implicitly to add a reference to the interface pointer.

See also AddRef
GetResultSetAccessor
IPB_ResultSetAccessor interface
ReleaseResultSetAccessor
CHAPTER 7    PBNI Interfaces, Structures, and Methods

FindClass
Description Searches for a class with a given name within a given group.
Syntax FindClass(pbgroup group, LPCTSTR name)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The handle of the group in which the class resides</td>
</tr>
<tr>
<td>name</td>
<td>The class name in lowercase</td>
</tr>
</tbody>
</table>

Return value pbclass or null on failure.
Examples This example finds the group associated with the f_getrow function and uses the group to find the class:

```c
    group = session->FindGroup("f_getrow", pbgroup_function);
    if ( group==NULL )
        return;
    cls = session->FindClass(group, "f_getrow");
    if ( cls==NULL )
        return;
```
Usage This method searches for a PowerBuilder class with the given name in the given group. For example, in a window definition w_1, w_1 is a group, and w_1 and controls contained in it are all classes of group w_1.
See also FindGroup
NewObject

FindClassByClassID
Description Searches for a class with a given name and a given ID.
Syntax FindClass(pbgroup group, pbint classID)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The handle of the group in which the class resides</td>
</tr>
<tr>
<td>classID</td>
<td>The class name in lowercase</td>
</tr>
</tbody>
</table>

Return value pbclass or null on failure.
Usage This method searches for a PowerBuilder class with the given name and the given ID.
See also FindGroup
NewObject
**FindGroup**

**Description**
Searches for a group with a given name and group type in the current library list.

**Syntax**
FindGroup(LPCTSTR name, pbgroup_type type)

**Argument** | **Description**
--- | ---
name | The group name in lowercase

**Return value**
pbgroup or null on failure.

**Examples**
This example finds the group associated with user_exception and uses the group to find the class:

```c
group = session->FindGroup("user_exception",
                         pbgroup_userobject);
if ( group==NULL )
    return;
cls = session->FindClass(group, "user_exception")
```

See also
FindClass
NewObject

**FindMatchingFunction**

**Description**
Finds a function that has the specified argument list.

**Syntax**
FindMatchingFunction(pbclass cls, LPCTSTR methodName, PBRoutineType rt, LPCTSTR readableSignature)

**Argument** | **Description**
--- | ---
cls | pbclass containing the method.
methodName | The string name of the method in lowercase.
rt | Type of the method: PBRT_FUNCTION for function or PBRT_EVENT for event.
readableSignature | A comma-separated string listing the types of the method’s arguments. The return type of the method is not included in the string. See the Usage section for examples.

**Return value**
pbmethodID.
Examples

This example returns the method ID of a function named `uf_test` that takes an integer and a double as arguments:

```cpp
pbclass cls;
pbmethodID mid;
PBCallInfo* ci = new PBCallInfo;
unsigned long ret_val;

cls = Session -> GetClass(myobj);
mid = Session -> FindMatchingFunction(cls, "uf_test",
PBRT_FUNCTION, "int, double");

Session -> InitCallInfo(cls, mid, ci);
```

Usage

`FindMatchingFunction` provides an alternative to the `GetMethodID` function. It requires a list of the function’s arguments (the `readableSignature`) instead of the signature obtained using the `pbsig125` tool.

This table shows the `readableSignature` for each of several functions.

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>void test1()</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>int test2()</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>string test3(int a, double b)</td>
<td>&quot;int, double&quot;</td>
</tr>
</tbody>
</table>
| datastore test4(powerobject a[],
    double b[2 to 10, 1 to 7]) | "powerobject[], double[2 to 10, 1 to 7]" |
| int test5(readonly int a[10,20],
    ref long c[]) | "readonly int[10,20],
    ref long[]" |

`FindMatchingFunction` does not check the access type of the function, so you can use it to obtain the method ID of a private function. `GetMethodID` cannot obtain the method ID of a private function.

See also

`GetMethodID`
FreeCallInfo

Description: Frees memory allocated by InitCallInfo.

Syntax: FreeCallInfo(PBCallInfo *ci)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ci</td>
<td>A pointer to the preallocated PBCallInfo structure</td>
</tr>
</tbody>
</table>

Return value: None.

Examples: FreeCallInfo should be called when the PBCallInfo structure is no longer needed:

```c
Session->InvokeObjectFunction(myobj, mid, ci);

ret_val = ci.returnValue->GetInt();
Session->FreeCallInfo(ci);
delete ci;
return ret_val;
```

Usage: This method frees memory allocated by InitCallInfo but does not free the structure ci itself.

See also: InitCallInfo

Get<type>ArrayItem

Description: Obtains the value of an array item of a specified type.

Syntax:

- GetBlobArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetBoolArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetByteArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetCharArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetDateArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetDateTimeArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetDecArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetDoubleArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetIntArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetLongArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetLongLongArrayItem (pbarray array, pblonglong dim[], pbboolean& IsNull)
- GetObjectArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
- GetRealArrayItem (pbarray array, pblong dim[], pbboolean& IsNull)
GetStringArrayItem ( pbarray array, pblong dim[], pbboolean& IsNull )
GetTimeArrayItem ( pbarray array, pblong dim[], pbboolean& IsNull )
GetUintArrayItem ( pbarray array, pblong dim[], pbboolean& IsNull )
GetUlongArrayItem ( pbarray array, pblong dim[], pbboolean& IsNull )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid pbarray structure</td>
</tr>
<tr>
<td>dim</td>
<td>The dimension of the array item to be obtained</td>
</tr>
<tr>
<td>IsNull</td>
<td>Indicates whether the array item is null</td>
</tr>
</tbody>
</table>

Return value
The value of the array item.

Examples
This example gets the value of an array item of type pbobject:

```cpp
pbobject pPBObject = NULL;
pbboolean bIsNull = 0;
pblong dim[1];

dim[0] = pbl + 1;
pPBObject = session->GetObjectArrayItem(array, dim,
                                       bIsNull);
```

See also
GetArrayInfo
GetArrayItemType
GetArrayLength
IsArrayItemNull
NewBoundedObjectArray
NewBoundedSimpleArray
NewUnboundedObjectArray
NewUnboundedSimpleArray
ReleaseArrayInfo
SetArrayItemToNull
SetArrayItemValue
Set<type>ArrayItem
IPB_Session interface

**Get<type>Field**

Description: A set of methods that gets the value of an instance field of an object.

Syntax:

- `GetArrayField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetBlobField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetBoolField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetByteField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetCharField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetDateField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetDateTimeField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetDecField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetDoubleField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetIntField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetLongField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetLongLongField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetObjectField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetRealField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetStringField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetTimeField (pobject obj, pbfieldID fid, pbint value)`
- `GetUintField (pobject obj, pbfieldID fid, pbboolean& isNull)`
- `GetUlongField (pobject obj, pbfieldID fid, pbboolean& isNull)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj</code></td>
<td>The handle of the object whose field is to be accessed</td>
</tr>
<tr>
<td><code>fid</code></td>
<td>The field ID of the specified object</td>
</tr>
<tr>
<td><code>isNull</code></td>
<td>Indicates whether the field is null</td>
</tr>
</tbody>
</table>

Return value: A predefined PBNI datatype that corresponds to the PowerBuilder datatype in the method name.

Examples:

This example gets the value of a field of type `pbstring`:

```c
pbboolean isNull;
pbstring pstr = session->GetStringField(proxy, fid, isNull);
if (pstr != NULL)
{
    myclass = session->GetString(pstr);
    // process myclass
```
Get\langle\text{type}\rangle\text{GlobalVar}

Description
A set of methods that gets the value of a global variable of a specific datatype.

Syntax
- GetArrayGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetBlobGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetBoolGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetByteGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetCharGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetDateGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetDateTimeGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetDecGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetDoubleGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetIntGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetLongGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetLongLongGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetObjectGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetRealGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetStringGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetTimeGlobalVar (pbfieldID \textit{fid}, pbint \textit{value})
- GetUintGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})
- GetUlongGlobalVar (pbfieldID \textit{fid}, pbboolean& \textit{isNull})

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{fid}</td>
<td>The field ID of the global variable</td>
</tr>
<tr>
<td>\textit{isNull}</td>
<td>Indicates whether the variable is null</td>
</tr>
</tbody>
</table>
IPB_Session interface

Return value
A predefined PBNI datatype that corresponds to the PowerBuilder datatype in the method name.

Examples
This code gets the value of a global variable of datatype long using its field ID:

```powerbuilder
fid = session -> GetGlobalVarID("l_gvar");
l_val = session -> GetLongGlobalVar(fid, isNull);
session -> SetLongGlobalVar(fid, l_val + 1);
```

See also
GetGlobalVarID
GetGlobalVarType
IsGlobalVarArray
IsGlobalVarNull
IsGlobalVarObject
SetGlobalVarToNull
Set<type>GlobalVar

Get<type>SharedVar

Description
A set of methods that gets the value of a shared variable of a specific datatype.

Syntax
```powerbuilder
GetArraySharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetBlobSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetBoolSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetByteSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetCharSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetDateSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetDateTimeSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetDecSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetDoubleSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetIntSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetLongSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetLongLongSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetObjectSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetRealSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetStringSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetTimeSharedVar ( pbgroup group, pbfieldID fid, pbint value )
GetUintSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetUlongSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
```
Return value
A predefined PNI datatype that corresponds to the PowerBuilder datatype in the method name.

Examples
This code gets the value of a shared variable of type integer:

```c
curGroup = session -> GetCurrGroup();
fid = session -> GetSharedVarID(curGroup, "i_svar");
if (fid == 0xffff)
{
    MessageBox(NULL, "Illegal fid!", "default", MB_OK);
    return;
}
_i_val = session->GetIntSharedVar(curGroup, fid, _isNull);
_i_val = session->SetIntSharedVar(curGroup, fid, _i_val+1);
```

See also
GetSharedVarID
GetSharedVarType
IsSharedVarArray
IsSharedVarNull
IsSharedVarObject
Set<type>SharedVar
SetSharedVarToNull
**GetArrayInfo**

**Description**: Obtains information about an array.

**Syntax**: `GetArrayInfo(parray array)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid array handle</td>
</tr>
</tbody>
</table>

**Return value**: `PBAArrayInfo*`.  

**Examples**: This IF-ELSE statement populates a PBAArrayInfo structure if the array in the first value of a PBCallInfo structure is not null:

```c
if ( !(ci->pArgs->GetAt(0)->IsNull()) )
{
    array = ci->pArgs->GetAt(0)->GetArray();
    pArrayInfo = session->GetArrayInfo (array);
    pArrayItemCount = session->GetArrayLength(array);
}
else
{
    // NULL array
    pArrayItemCount = 0;
}
```

**Usage**: If the array is an unbounded array, the bounds information in PBAArrayInfo is undetermined. The returned PBAArrayInfo must be freed later by `ReleaseArrayInfo`.

**See also**:  
- `Get<type>ArrayItem`
- `GetArrayItemType`
- `GetArrayLength`
- `IsArrayItemNull`
- `NewBoundedObjectArray`
- `NewBoundedSimpleArray`
- `NewUnboundedObjectArray`
- `NewUnboundedSimpleArray`
- `ReleaseArrayInfo`
- `SetArrayItemToNull`
- `SetArrayItemValue`
- `Set<type>ArrayItem`
#### GetArrayItemType

**Description**
Obtains the datatype of an item in an array.

**Syntax**
GetArrayItemType( pbarray array, pblong dim[ ])

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid pbarray structure.</td>
</tr>
<tr>
<td>dim</td>
<td>A pblong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of array.</td>
</tr>
</tbody>
</table>

**Return value**
pbuint.

**See also**
Get<type>ArrayItem
GetArrayInfo
GetArrayLength
IsArrayItemNull
NewBoundedObjectArray
NewBoundedSimpleArray
NewUnboundedObjectArray
NewUnboundedSimpleArray
ReleaseArrayInfo
SetArrayItemToNull
SetArrayItemValue
Set<type>ArrayItem
IPB_Session interface

**GetArrayLength**

Description

Obtains the length of an array.

Syntax

GetArrayLength(parray array)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid array handle</td>
</tr>
</tbody>
</table>

Return value

pblong.

Examples

This IF-ELSE statement populates a PBArryInfo structure. If the array in the first value of a PBCallInfo structure is not null, it sets the value of the pArrayItemCount variable to the length of the array:

```c
if ( !(ci->pArgs->GetAt(0)->IsNull()) )
{
    array = ci->pArgs->GetAt(0)->GetArray();
    pArrayInfo = session->GetArrayInfo(array);
    pArrayItemCount = session->GetArrayLength(array);
}
else
{
    // NULL array
    pArrayItemCount = 0;
}
```

See also

Get<type>ArrayItem
GetArrayInfo
IsArrayItemNull
NewBoundedObjectArray
NewBoundedSimpleArray
NewUnboundedObjectArray
NewUnboundedSimpleArray
ReleaseArrayInfo
SetArrayItemToNull
SetArrayItemValue
Set<type>ArrayItem
GetBlob

**Description**
Returns a pointer to the data buffer for a blob.

**Syntax**
GetBlob(pbblob *bin)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>bin</em></td>
<td>A pointer to the source buffer</td>
</tr>
</tbody>
</table>

**Return value**
*void*.

**Examples**
In this **CASE** clause, the value returned from GetBlob is cast to the **LPCTSTR** variable *pStr*. If it is not **null**, the return value in the **PBCallInfo** structure is set to the value of the blob:

```c
    case pbvalue_blob:
        pStr = (LPCTSTR)Session->GetBlob(retVal.blob_val);
        if (strncmp(pStr, "null", 4)==0 )
            ci -> returnValue ->SetToNull();
        else
            {
                ci -> returnValue->SetBlob(retVal.blob_val);
                Session -> ReleaseValue(retVal);
            }
        break;
```

**See also**
GetBlobLength
NewBlob
SetBlob
### GetBlobLength

**Description**

Returns the length in bytes of blob data in a buffer.

**Syntax**

```
GetBlobLength (pbblob bin)
```

**Argument** | **Description**
--- | ---
bin | A pointer to the source buffer

**Return value**

pblong.

**Examples**

In this example, the IPB_Value GetBlob function is used to get a blob value from the PBCallInfo structure. The length of the blob is used as an argument to the NewBlob function:

```c
PBCallInfo* ci = new PBCallInfo;
pbblob ret_val;
pblong bloblen;

ret_val = ci.returnValue->GetBlob();
bloblen = Session->GetBlobLength(ret_val);
ret_val = Session->NewBlob((Session->GetBlob(ret_val), bloblen);
```

**See also**

- GetBlob
- NewBlob
- SetBlob
GetClass

Description
Returns the class handle of a PowerBuilder object. This function is most frequently used to obtain a class handle for use with the GetMethodID function.

Syntax
GetClass (pboject obj)

Argument | Description
--- | ---
obj | A valid PowerBuilder object handle

Return value
pbclass or null on error.

Examples
In this example, GetClass is used to obtain the class of a variable of type UserData so that the class can be used as an argument to the GetMethodID function:

```cpp
BOOL CALLBACK CFontEnumerator::EnumFontProc

LPLOGFONT lplf,
LPNEWTEXTMETRIC lpntm,
DWORD FontType,
LPVOID userData

UserData* ud = (UserData*)userData;
pbclass clz = ud->session->GetClass(ud->object);
pbmethodID mid = ud->session->GetMethodID
(clz, "onnewfont", PBRT_EVENT, "IS");

PBCallInfo ci;
ud->session->InitCallInfo(clz, mid, &ci);

pbstring str = ud->session->NewString
(lplf->lfFaceName);
ct.pArgs->GetAt(0)->SetPBString(str);
ud->session->TriggerEvent(ud->object, mid, &ci);
pbint ret = ci.returnValue->GetInt();
ud->session->FreeCallInfo(&ci);

return ret == 1 ? TRUE : FALSE;
```

See also
GetClassName
GetMethodID
**GetClassName**

**Description**
Returns the name of a class in lowercase.

**Syntax**
GetClassName(pbclass cls)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle</td>
</tr>
</tbody>
</table>

**Return value**
LPCTSTR.

**Examples**
This example gets the name of a class and sets the size of the variable stLength to the length of the returned string plus 1:

```c
LPCTSTR myClassName = session->GetClassName( myClass );
size_t stLength = strlen( (LPSTR)myClassName ) + 1;
```

**Usage**
When you have finished using the name, call the ReleaseString method to free the memory acquired.

**See also**
GetClass
ReleaseString

---

**GetCurrGroup**

**Description**
Obtains the name of the current group.

**Syntax**
GetCurrGroup( )

**Return value**
pbgroup or null on failure.

**Examples**
This example gets the name of the current group and uses it to obtain the identifier of a shared variable, get the shared variable’s value, and reset the shared variable’s value:

```c
curGroup = session -> GetCurrGroup();
fid = session -> GetSharedVarID(curGroup, "i_svar");
if (fid == 0xffff)
{
    MessageBox(NULL, "Illegal fid!", "default", MB_OK);
    return;
}
  i_val = session->GetIntSharedVar(curGroup, fid, isNull);
  session-> SetIntSharedVar(curGroup, fid, i_val+1);
```

**See also**
Get<type>SharedVar
GetSharedVarID
Set<type>SharedVar
**GetDateString**

Description: Converts data in a `pbdate` object to a string.

Syntax: `GetDateString(pbdate date)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date</code></td>
<td>The <code>pbdate</code> data object to be converted to a string.</td>
</tr>
</tbody>
</table>

Return value: LPCTSTR.

See also: NewDate, ReleaseDateString, SetDate

**GetDateTimeString**

Description: Converts data in a `pbdatetime` object to a string.

Syntax: `GetDateTimeString(pbdatetime datetime)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datetime</code></td>
<td>The <code>pbdatetime</code> data object to be converted to a string.</td>
</tr>
</tbody>
</table>

Return value: LPCTSTR.

See also: NewDateTime, ReleaseDateTimeString, SetDateTime
**GetDecimalString**

**Description**
Converts decimal data in a pbdec object to a string.

**Syntax**
GetDecimalString(pbdec dec)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec</td>
<td>The pbdec data object to be converted to a string.</td>
</tr>
</tbody>
</table>

**Return value**
LPCTSTR.

**Examples**
This code checks whether a value in the PBCallInfo structure is null. If it is not, it sets the value in the pArguments array to the value in PBCallInfo:

```c
case pbvalue_dec:
    if (ci->pArgs->GetAt(i)->IsNull())
    {
        pArguments[i].dec_val = Session->NewDecimal();
        Session->SetDecimal(pArguments[i].dec_val, "1.0");
    }
    else
        pArguments[i].dec_val = ci->pArgs->GetAt(i)->GetDecimalString();
    break;
```

**See also**
NewDecimal
ReleaseDecimalString
SetDecimal

---

**GetEnumItemName**

**Description**
Obtains the name of an enumerated variable.

**Syntax**
GetEnumItemName(LPCTSTR enumName, long enumItemValue)

**Return value**
LPCTSTR.

**Usage**
When you have finished using the name, call the ReleaseString method to free the memory acquired.

**See also**
GetEnumItemValue
ReleaseString
GetEnumItemValue

Description
Obtains the value of an enumerated variable.

Syntax
GetEnumItemValue(LPCTSTR enumName, LPCTSTR enumItemName)

Return value
Long.

Examples
This example gets the numeric value for the boolean! enumerated value, then uses it to return the string value:

```plaintext
pblong lType = session->GetEnumItemValue("object", boolean" ); // returns 138
LPCTSTR szEnum = session->GetEnumItemName("object", lType ); // returns "boolean"
```

Usage
GetEnumItemValue and GetEnumItemName support enumerated types. They allow you to convert the name of an enumerated value, a string with an appended exclamation mark (!), to an integer value, and vice versa.

The ! character must be omitted
When you use these functions, the enumItemName should not use the appended exclamation mark (!) character.

To return an enumerated value from an extension to PowerScript, you must use the SetLong function to set the value of the enumerated variable into IPB_Value. Using SetInt or SetShort fails. However, you can use GetInt or GetShort as well as GetLong to obtain the enumerated variable's value, assuming the value is in the appropriate range. For example, if you attempt to use GetInt to obtain a value that is more than 32767, the returned value is truncated.

See also
GetEnumItemName
IPB_Session interface

GetException

Description Obtains the current thrown exception object.

Syntax GetException ()

Return value pboject.

Examples This code gets the current exception object, clears the exception, and gets the class of the exception object:

```cpp
pbclass cls;
pobject ex;
...
ex = session->GetException();
session->ClearException();
cls = session->GetClass(ex);
```

See also ClearException
HasExceptionThrown

GetFieldID

Description Obtains the internal ID of a class instance variable.

Syntax GetFieldID(pbclass cls, LPCTSTR fieldName)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>The class in which the field resides</td>
</tr>
<tr>
<td>fieldName</td>
<td>The instance member name, in lowercase</td>
</tr>
</tbody>
</table>

Return value pbfieldID or 0xffff if a field ID cannot be found.

Examples This function obtains the identifier of a class’s visible field, if it exists, and uses it to set the value of the field:

```cpp
void CallBack::f_setvisible(IPB_Session* session, pboject dwobj)
{
    pbclass cls;
    IPB_Value* pv;
    pbfieldID fid;
    pbstring strtmp;
    bool isTrue;
    pbool isNull;

    cls = session->GetClass(dwobj);
    fid = session->GetFieldID(cls, "visible");
```
if (fid == kUndefinedFieldID)
    return;
isTrue = session->GetBoolField(dwobj, fid, isNull);
if (isTrue)
    session->SetBoolField(dwobj, fid, false);
else
    session->SetBoolField(dwobj, fid, true);
return;
}

Usage
GetFieldID is one of a set of functions that allows native code to access the
fields of Java objects and get and set their values. You use GetFieldID to retrieve
the value of a field, specifying the class name and the field name. The field ID
returned can be used as an argument to the related functions.

See also
GetFieldID
Get.Field
GetNumFields
IsFieldArray
IsFieldNull
IsFieldObject
Set.Field
SetFieldToNull

GetFieldName
Description
Obtains the name of the specified field.
Syntax
GetFieldName(pbclass cls, pbfieldID fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>The class that defines the field</td>
</tr>
<tr>
<td>fid</td>
<td>The internal ID of the class instance variable</td>
</tr>
</tbody>
</table>

Return value
LPCTSTR. The field name of the specified field. If an incorrect field ID is
specified, this function returns null.

Usage
When you have finished using the name, call the ReleaseString method to free
the memory acquired.

See also
GetFieldID
ReleaseString
**GetFieldType**

**Description**
Obtains the datatype of a field declared by a class.

**Syntax**
GetFieldType(pbclass cls, pffieldID fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>The class that defines the field</td>
</tr>
<tr>
<td>fid</td>
<td>The internal ID of the class instance variable</td>
</tr>
</tbody>
</table>

**Return value**
pbint. A simple datatype defined in the list of pbvalue_type enumerated types, such as pbvalue_int. See “PBNI enumerated types” on page 84.

**Examples**
This statement gets the type of the specified field ID:

```cpp
pbint pbfieldType = session->GetFieldType(cls, fid);
```

**See also**
GetFieldID
Get<type>Field
GetNumOfFields
IsFieldArray
IsFieldNull
IsFieldObject
Set<type>Field
SetFieldToNull

**GetGlobalVarID**

**Description**
Returns the internal ID of a global variable.

**Syntax**
GetGlobalVarID(LPCTSTR name)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the global variable in lowercase</td>
</tr>
</tbody>
</table>

**Return value**
pffieldID or null on failure.

**Examples**
This example gets the internal identifier of a long variable and uses it to get and set a global variable:

```cpp
fid = session -> GetGlobalVarID("l_gvar");
1_val = session -> GetLongGlobalVar(fid, isNull);
session -> SetLongGlobalVar(fid, 1_val + 1);
```

**See also**
GetGlobalVarType
Get<type>GlobalVar
IsGlobalVarArray
IsGlobalVarNull
IsGlobalVarObject
SetGlobalVarToNull
Set<type>GlobalVar

GetGlobalVarType
Description Obtains the datatype of a global variable.
Syntax GetGlobalVarType(pbfieldID \( fid \))

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( fid )</td>
<td>The internal ID of the class instance variable</td>
</tr>
</tbody>
</table>

Return value pbuint. A simple datatype defined in the list of pbvalue_type enumerated types.

Examples This code tests getting and setting a global integer variable using the field ID \( fid \):

```c++
    fid = session -> GetGlobalVarID("i_gvar");
    if (session -> GetGlobalVarType(fid) == pbvalue_int)
    {
        i_val = session -> GetIntGlobalVar(fid,isNull);
        session -> SetIntGlobalVar(fid,i_val+1);
    }
```

See also GetGlobalVarID
Get<type>GlobalVar
IsGlobalVarArray
IsGlobalVarNull
IsGlobalVarObject
SetGlobalVarToNull
Set<type>GlobalVar
IPB_Session interface

GetMarshaler

Description
Obtains the marshaler object associated with a proxy object.

Syntax
GetMarshaler(pbproxyObject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>An object of type pbproxyObject for which you want to find the marshaler.</td>
</tr>
</tbody>
</table>

Return value
IPBX.Marshaler*.

Examples
This code creates a Java marshaler object and associates it with a proxy. Later, GetMarshaler is used to get the marshaler object:

```c
// Create JavaMarshaler
JavaMarshaler* marshaler = new JavaMarshaler(env, proxy, job);

// Associate the JavaMarshaler with the // PowerBuilder proxy
session->SetMarshaler(proxy, marshaler);

ci->pArgs->GetAt(0)->SetObject(proxy);

ci->returnValue->SetLong(kSuccessful);

return PBX_OK;

// Get the marshaler
IPBX.Marshaler* pIPBX.Marshaler = NULL;

pIPBX.Marshaler = (IPBX.Marshaler*)session->GetMarshaler(proxy);
```

See also
SetMarshaler
GetMethodID

Description
Returns the ID of the requested method.

Syntax
GetMethodID(pbclass cls, LPCTSTR methodName, PBRoutineType rt, LPCTSTR signature, pbboolean publicOnly)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>pbclass containing the function.</td>
</tr>
<tr>
<td>methodName</td>
<td>The string name of the method in lowercase.</td>
</tr>
<tr>
<td>rt</td>
<td>Type of the method: PBRT_FUNCTION for function or PBRT_EVENT for event.</td>
</tr>
<tr>
<td>signature</td>
<td>Internal signature of the PowerBuilder function, used to identify polymorphic methods in one class. Obtained with the pbsig125 tool. If the signature is a null string (&quot;&quot;), the first method found with the name methodName is returned.</td>
</tr>
<tr>
<td>publicOnly</td>
<td>A boolean that determines whether only public methods are searched (true) or all methods are searched (false). The default is true.</td>
</tr>
</tbody>
</table>

Return value
pbMethodID of the method or kUndefinedMethodID on error.

Examples
This function uses GetMethodID to obtain the identifier (mid) of the onnewfont function so that the identifier can be used to initialize the PBCallInfo structure and call the function:

```c
BOOL CALLBACK CFontEnumerator::EnumFontProc
(
    LPLOGFONT lplf,
    LPNEWTEXTMETRIC lpntm,
    DWORD FontType,
    LPVOID userData
)
{
    UserData* ud = (UserData*)userData;
    pbclass clz = ud->session->GetClass(ud->object);
    pbmethodID mid = ud->session->GetMethodID(clz, "onnewfont", PBRT_EVENT, "IS");

    PBCallInfo ci;
    ud->session->InitCallInfo(clz, mid, &ci);
    pbstring str = ud->session->NewString(lplf->lfFaceName);
    ci.pArgs->GetAt(0)->SetPBString(str);
    ud->session->TriggerEvent(ud->object, mid, &ci);
    pbint ret = ci.returnValue->GetInt();
```
IPB_Session interface

```c
ud->session->FreeCallInfo(&ci);

return ret == 1 ? TRUE : FALSE;
```

Usage

The `GetMethodID` function is used to obtain the ID of a method so you can use it to invoke functions and trigger events.

See also

- `FindMatchingFunction`
- `InvokeObjectFunction`
- `TriggerEvent`
- "Calling PowerScript from an extension" on page 39

### GetMethodIDByEventID

**Description**

Returns the ID of the method that has a given predefined PowerBuilder event ID.

**Syntax**

```c
GetMethodIDByEventID(pbclass cls, LPCTSTR eventID)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cls</code></td>
<td>pbclass containing the method</td>
</tr>
<tr>
<td><code>eventID</code></td>
<td>A PowerBuilder predefined event string, such as pbm_bnclicked</td>
</tr>
</tbody>
</table>

**Return value**

`pbMethodID` of the method or `kUndefinedMethodID` on error.

**Examples**

This statement obtains the ID of the event identified by the name `pbm_lbuttonup`:

```c
pbmethodID mid = d_session->GetMethodIDByEventID(clz, "pbm_lbuttonup");
```

See also

- `GetMethodID`
GetNativeInterface

**Description**
Obtains a pointer to the interface of a native class.

**Syntax**
```
GetNativeInterface(pbobject obj)
```

**Argument** | **Description**
--- | ---
obj | A valid object handle

**Return value**
IPBX_UserObject.

**Examples**
This example invokes the function _f_retrieve_ in the native class `Cmy_pbni` to retrieve a DataWindow object:

```c
long f_retrieve(IPB_Session* session, pbint iarg, pbobject dwObj, pbobject extObj)
{
    Imy_pbni* pImy_pbni = NULL;
pblong lRet;
if (session -> IsNativeObject(extObj) )
{
    pImy_pbni = (Imy_pbni*) session -> GetNativeInterface(extObj);
    lRet = pImy_pbni-> f_Retrieve(session, iarg, dwObj);
}
return lRet;
}
```

**Usage**
Use this method in conjunction with _IsNativeObject_ to obtain a direct reference to the IPBX_UserObject associated with a native class in the same PowerBuilder extension. The class and its methods can then be accessed directly.

**See also**
_IsNativeObject_
GetNumOfFields

Description
Returns the number of fields in the specified class.

Syntax
GetNumOfFields(pbclass cls)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle for the class whose field is to be accessed</td>
</tr>
</tbody>
</table>

Return value
pbulong.

Examples
This code gets the numbers of fields in the class clz:

```powershell
pbclass clz = d_session->GetClass(d_pbobj);
pulong nf = d_session->GetNumOfFields(clz);
```

See also
GetFieldID
Get<type>Field
IsFieldArray
IsFieldNull
IsFieldObject
SetFieldToNull
Set<type>Field

GetPBAAnyArrayItem

Description
Obtains the value of a global variable of type Any.

Syntax
GetPBAAnyArrayItem( parray array, pbulong dim[], pbboolean& isNull )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid parray structure.</td>
</tr>
<tr>
<td>dim</td>
<td>A pbulong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of array.</td>
</tr>
<tr>
<td>isNull</td>
<td>Indicates whether the variable is null</td>
</tr>
</tbody>
</table>

Return value
IPB_Value*.

Usage
See GetPBAAnyField.

See also
GetPBAAnyField
GetPBAAnyGlobalVar
GetPBAAnySharedVar
GetPBAnyField

Description

Obtains the value of a variable of type Any.

Syntax

GetPBAnyField( pbobject obj, pbfieldID fid, pbboolean& isNull )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid object handle for the object whose value is to be obtained</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the variable</td>
</tr>
<tr>
<td>isNull</td>
<td>Indicates whether the variable is null</td>
</tr>
</tbody>
</table>

Return value

IPB_Value*.

Examples

This example tests all the functions used to get the value of variables of type Any, using PushLocalFrame and PopLocalFrame to simulate the scope of a function call:

```c
session->PushLocalFrame();
pbgroup vgroup = session->FindGroup("n_test",    
   pbgroup_userobject);
pbclass vcls = session->FindClass(vgroup, "n_test");
pbobject vobj = session->NewObject(vcls);
pbboolean isNull;

pbfieldID vfid = session->GetFieldID(vcls, "i_a");
IPB_Value* value = session->GetPBAnyField(vobj, 
   vfid, isNull);
pbstring str = value->GetString(); // save actual value

vfid = session->GetSharedVarID(vgroup, "s_a");
value = session->GetPBAnySharedVar(vgroup, 
   vfid, isNull);
//Get the actual value here.

vfid = session->GetGlobalVarID("g_a");
value = session->GetPBAnyGlobalVar(vfid, isNull);   
//Get the actual value here.

vfid = session->GetFieldID(vcls, "i_array");
pbarray arr = session->GetArrayField(vobj, 
   vfid, isNull);  //Get the any array first.

long dim = 1;
value = session->GetPBAnyArrayItem(arr, &dim, isNull); //Get the actual value here.

session->PopLocalFrame();
```
Usage

The value you retrieve must be of datatype Any to use this function; that is, the variable associated with the function must be declared as a variable of type Any in the development environment. If it is not, the function returns a null pointer and the value of isNull is set to true.

This function returns a pointer to an IPB_Value instance. When it is called, memory is allocated for the returned IPB_Value instance, and the pointer is recorded in the current local frame. The pointer is deleted automatically when the current local frame is popped, which occurs when the current local function returns (you can also call PopLocalFrame to force the frame to be popped).

If you want to use the value returned, you must save the value pointed to by the IPB_Value instance (not the IPB_Value instance itself) before the frame is popped. If you save the pointer itself, the value is only valid until the original value is destroyed.

You can use the AcquireValue function to save the value, or one of the IPB_Value Get<type> functions. For example, the following code saves the string value in the IPB_Value instance ivalue into the string str. The value in str can be used after the local frame is popped and ivalue is deleted:

```c
IPB_Value* ivalue = session->GetPBAnyField(vobj, vfid, isNull);
    pbstring str = ivalue->GetString();
```

If you do not know the actual datatype of the Any variable, use the IPB_Value GetType function to get its datatype first, then use the appropriate get function to get its value.

---

**IPB_Value holds a reference to the original value**

The value in the IPB_Value instance is a reference to the original value. If you change the actual value of the returned IPB_Value, the original value is also changed. If you use the AcquireValue function to save the value, it clones a new IPB_Value and resets the existing IPB_Value pointer.

---

See also

GetPBAnyArrayItem
GetPBAnyGlobalVar
GetPBAnySharedVar
CHAPTER 7  
PNI Interfaces, Structures, and Methods

GetPBAnyGlobalVar
Description Obtains the value of a global variable of type Any.
Syntax GetPBAnyGlobalVar( pbfieldID fid, pbboolean& isNull )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fid</td>
<td>The field ID of the variable</td>
</tr>
<tr>
<td>isNull</td>
<td>Indicates whether the variable is null</td>
</tr>
</tbody>
</table>

Return value IPB_Value*.
Usage See GetPBAnyField.
See also GetPBAnyArrayItem
GetPBAnyField
GetPBAnySharedVar

GetPBAnySharedVar
Description Obtains the value of a shared variable of type Any.
Syntax GetPBAnySharedVar( pbgroup group, pbfieldID fid, pbboolean& isNull )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group to which the variable belongs</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the variable</td>
</tr>
<tr>
<td>isNull</td>
<td>Indicates whether the variable is null</td>
</tr>
</tbody>
</table>

Return value IPB_Value*.
Usage See GetPBAnyField.
See also GetPBAnyArrayItem
GetPBAnyField
GetPBAnyGlobalVar
**GetProp**

Description: Retrieves a pointer to the data value of a variable that has been registered as a shared property for the current IPB session.

Syntax: `GetProp(LPCTSTR name)`

**Argument** | **Description**
--- | ---
*name* | The name of the variable whose value is to be retrieved.

Return value: `Void*`. If the variable does not exist, returns `null`.

Examples: See `SetProp`.

Usage: The variable’s name must first be registered with the session using the `SetProp` function.

See also: `RemoveProp`  
`SetProp`  

**GetResultSetAccessor**

Description: Obtains an interface through which you can read data from a result set.

Syntax: `GetResultSetAccessor (pbobject rs)`

**Argument** | **Description**
--- | ---
*rs* | A `pbobject` holding a result set obtained using `CreateResultSet`.

Return value: `IPB_ResultSetAccessor`.

Examples: This example gets a result set, *rs*, from the return value of a PowerScript function and uses it to create an `IPB_ResultSetAccessor` object, *rsa*:

```c
pbobject rs = ci.returnValue->GetObject();  
IPB_ResultSetAccessor* rsa = session->GetResultSetAccessor(rs);  
```

See also: `CreateResultSet`  
`ReleaseResultSetAccessor`
GetSharedVarID

**Description**

Returns the internal ID of a shared variable.

**Syntax**

```
GetSharedVarID(pbgroup group, LPCTSTR fieldname)
```

**Argument** | **Description**
--- | ---
`group` | The group to which the shared variable belongs
`fieldname` | The name of the field that contains the shared variable, in lowercase

**Return value**

`pbfieldID`. Returns `0xffff` if the ID cannot be found.

**Examples**

This code uses `GetSharedVarID` to obtain the field ID of a shared variable, then uses that ID to obtain the value of the variable:

```c
curGroup = session -> GetCurrGroup();
fid = session -> GetSharedVarID(curGroup,"i_svar");
if (fid == 0xffff)
{
    MessageBox(NULL, "Illegal fid!", "default", MB_OK);
    return;
}
i_val = session -> GetIntSharedVar(curGroup, fid,
    isNull);
```

**See also**

- `GetSharedVar`  
- `GetSharedVarType`  
- `IsSharedVarArray`  
- `IsSharedVarNull`  
- `IsSharedVarObject`  
- `Get<type>SharedVar`  
- `Set<type>SharedVar`  
- `SetSharedVarToNull`
IPB_Session interface

GetSharedVarType

Description
Obtains the datatype of the specified shared variable.

Syntax
GetSharedVarType ( pbgroup group, pbfieldID fid )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group to which the shared variable belongs</td>
</tr>
<tr>
<td>fid</td>
<td>The internal field ID of the shared variable</td>
</tr>
</tbody>
</table>

Return value
pbuint. A simple datatype defined in the list of pbvalue_type enumerated types.

Examples
This example gets the field ID of a shared variable, then uses that ID to get the type of the shared variable:

```
pbuint pbvaltype;
curGroup = session -> GetCurrGroup();
fid = session -> GetSharedVarID(curGroup,"i_svar");
pbvaltype = session -> GetSharedVarType(curGroup, fid);
```

See also
Get<type>SharedVar
GetSharedVarID
IsSharedVarArray
IsSharedVarNull
IsSharedVarObject
Set<type>SharedVar
SetSharedVarToNull
**GetString**

**Description**
Returns a pointer to the string passed in as an argument.

**Syntax**
GetString (pbstring* string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A pointer to a pbstring</td>
</tr>
</tbody>
</table>

**Return value**
LPCTSTR.

**Examples**
This example uses the IPB_Value GetString function to obtain a string value from the PBCallInfo structure. If the string is not null, the IPB_Session GetString function sets the value of the proxyname string to a pointer to the returned value:

```cpp
string proxyName;
{
    pbstring pn = ci->pArgs->GetAt(2)->GetString();

    if (pn == NULL)
    {
        ci->returnValue->SetLong(kInvalidProxyName);
        return PBX_OK;
    }
    else
    {
        proxyName = session->GetString(pn);
    }
}
```

**Usage**
When you have finished using the string, call the ReleaseString method to free the memory acquired.

**See also**
GetStringLength
NewString
ReleaseString
SetString
**GetStringLength**

**Description**
Returns the length of a string in bytes without the terminator.

**Syntax**
GetStringLength (pbstring string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The pbstring whose length is to be determined</td>
</tr>
</tbody>
</table>

**Return value**
pblong.

**Examples**
These statements set the value of a pblong variable to the length of a string:

```c
pblong long_val;
pbstring str_val;
long_val = session->GetStringLength(str_val);
```

**See also**
GetString, NewString, SetString

**GetSuperClass**

**Description**
Returns the ancestor class of the specified class, if any.

**Syntax**
GetSuperClass(pbclass cls)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle for the descendent class</td>
</tr>
</tbody>
</table>

**Return value**
pbclass or 0 if the class has no ancestor.

**Examples**
These statements get the class of an object in the PBCallInfo structure, the ancestor class of that class, and then the name of the ancestor class:

```c
pbclass cls, cls_parent;
LPCSTR clsname;

cls = Session->GetClass(ci->pArgs->GetAt(0)->GetObject());
cls_parent = Session->GetSuperClass(cls);
clsname = Session->GetClassName(cls_parent);
```

**See also**
GetClass, GetClassName
CHAPTER 7  PBNI Interfaces, Structures, and Methods

GetSystemClass

Description  Returns the first system class that the input class inherits from.

Syntax  GetSystemClass (pbclass cls)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A descendent class whose ancestor system class is to be determined</td>
</tr>
</tbody>
</table>

Return value  pbclass or null on error.

See also  GetMethodID

GetSystemGroup

Description  Returns a PowerBuilder internal system group.

Syntax  GetSystemGroup()

Return value  pbclass or null on error.

Usage  GetSystemGroup returns the PowerBuilder internal system group, which contains all the system types such as PowerObject, NonVisualObject, Structure, Window, CommandButton, and so on. You can use this system group to obtain a system class. You might need to call PowerScript functions in the PowerBuilder extension. To achieve this, you first need to get the pbclass that the PowerScript function class resides in. This code gets the PowerBuilder system function class:

```plaintext
pbgroup sysGroup = session->GetSystemGroup();
pbclass sysFuncClass = session->FindClass(sysGroup, "SystemFunctions");
```

After you get the system class, you can obtain the method ID of a PowerScript function by calling FindMatchingFunction, and then you can invoke the PowerScript function.

See also  FindMatchingFunction

GetSystemClass
IPB_Session interface

GetTimeString
Description Converts data in a pbtime object to a string.
Syntax GetTimeString(pbtime time)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>The pbtime data object to be converted to a string.</td>
</tr>
</tbody>
</table>

Return value LPCTSTR.
See also NewString
ReleaseTimeString
SetString

HasExceptionThrown
Description Checks for the existence of an exception that has been thrown but not cleared.
Syntax HasExceptionThrown()
Return value pbboolean. Returns true if a PowerBuilder exception has been thrown but not cleared.

Examples This example tests whether an exception has been thrown so it can be handled and cleared:

```powerbuilder
try {
    session->InvokeObjectFunction(pbobj, mid, &ci);
    // Was PB exception thrown?
    if (session->HasExceptionThrown()) {
        // Handle PB exception
        session->ClearException();
    }
}
```

See also ClearException
GetException
ThrowException
HasPBVisualObject

Description
Determines whether any PowerBuilder windows, visible or hidden, are still in existence.

Syntax
HasPBVisualObject()

Return value pbboolean. Returns true if any PowerBuilder windows are still alive. If any windows that are not response windows are still alive, the PowerBuilder application returns immediately unless you manually add a message loop.

Examples
This example is similar to the example for RestartRequested, but it includes a call to HasPBVisualObject that opens a message loop if the return value is true:

```c
PBXRESULT PB_MyWinAppRunner::RunApplication()
{
    PBXRESULT res;
    pbboolean restart = FALSE;

    do
    {
        res = StartApplication();
        if (res == PBX_OK)
            // Process message dispatch
            {
                MSG msg;
                while (GetMessage(&msg, 0, 0, 0))
                    {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);

                        if (!GetSession()->HasPBVisualObject())
                            break;
                    }
            }
        else
            break;

        restart = GetSession()->RestartRequested();
        if (restart)
            RecreateSession();
    } while (restart);

    return CleanApplication();
}
```
**IPB_Session interface**

Usage

RestartRequested and HasVisualPBObobject are used in the implementation of the IPB_VM RunApplication function. You no longer need to use an external message loop to check for Windows messages when you call the RunApplication function as you did in versions of PBNI prior to PowerBuilder 10.5.

See also

RestartRequested

RunApplication

**InitCallInfo**

Description

Initializes the PBCallInfo structure.

Syntax

InitCallInfo(pbclass cls, pbmethodID mid, PBCallInfo *ci)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>The pbclass containing the method</td>
</tr>
<tr>
<td>mid</td>
<td>The pbMethodID returned by GetMethodID</td>
</tr>
<tr>
<td>ci</td>
<td>A pointer to a preallocated PBCallInfo structure</td>
</tr>
</tbody>
</table>

Return value

PBXRESULT. Returns PBX_OK on success, and PBX_E_INVALID_ARGUMENT on failure.

Examples

This example shows the implementation of a TriggerEvent function in a visual class. It takes an event name as an argument, obtains the class and method ID needed to initialize the PBCallInfo structure, triggers the event, and frees the PBCallInfo structure:

```c
void CVisualExt::TriggerEvent(LPCTSTR eventName)
{
    pbclass clz = d_session->GetClass(d_pbobj);
    pbmethodID mid = d_session->GetMethodID(clz, eventName, PBRT_EVENT, "I");

    PBCallInfo ci;
    d_session->InitCallInfo(clz, mid, &ci);
    d_session->TriggerEvent(d_pbobj, mid, &ci);
    d_session->FreeCallInfo(&ci);
}
```

Usage

On return, this method allocates enough space for the arguments, and then initializes the arguments and return value. You must set appropriate values in the PBCallInfo structure. Note that the structure itself must have been allocated before the call.

See also

FreeCallInfo
InvokeClassFunction

Description
Invokes system or user global functions.

Syntax
InvokeClassFunction(pbclass cls, pbmethodID mid, PBCallInfo *ci)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>The class that contains the global function. If this is a system function, cls is obtained with GetSystemFunctionsClass; otherwise, it is obtained with FindGroup and FindClass, with the function name as the group/class name.</td>
</tr>
<tr>
<td>mid</td>
<td>The pbMethodID returned by GetMethodID.</td>
</tr>
<tr>
<td>ci</td>
<td>A pointer to a preallocated PBCallInfo structure.</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. Returns PBX_OK for success, or one of the following for failure:
- PBX_E_INVALID_ARGUMENT
- PBX_E_INVOKE_METHOD_INACCESSIBLE
- PBX_E_INVOKE_WRONG_NUM_ARGS
- PBX_E_INVOKE_REFARG_ERROR
- PBX_E_INVOKE_METHOD_AMBIGUOUS
- PBX_E_INVOKE_FAILURE
- PBX_E_INVOKE_FAILURE

Examples
This example gets the PowerBuilder system class and uses it to invoke the double function:

```c
cls = session->GetSystemClass();
mid = session->GetMethodID
   (cls, "double", PBRT_FUNCTION, "DA");
session->InitCallInfo(cls, mid, ci);
    ci->pArgs -> GetAt(0) -> SetPBString(mystr);
session -> InvokeClassFunction(cls, mid, ci);
```

Usage
On return, this method allocates enough spaces for the arguments, and then initializes arguments and return value. You must set appropriate values in the PBCallInfo structure. Note that the structure itself must have been allocated before the call.

See also
InvokeObjectFunction
IPB_Session interface

InvokeObjectFunction

Description
Invokes a class member method.

Syntax
InvokeObjectFunction(pbobject obj, pbmethodID mid, PBCallInfo *ci)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The pbobject containing the method</td>
</tr>
<tr>
<td>mid</td>
<td>The pbMethodID returned by GetMethodID</td>
</tr>
<tr>
<td>ci</td>
<td>A pointer to a preallocated PBCallInfo structure</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. Returns PBX_OK for success, or one of the following for failure:
- PBX_E_INVALID_ARGUMENT
- PBX_E_INVOKE_METHOD_INACCESSIBLE
- PBX_E_INVOKE_WRONG_NUM_ARGS
- PBX_E_INVOKE_REFARG_ERROR
- PBX_E_INVOKE_METHOD_AMBIGUOUS
- PBX_E_INVOKE_FAILURE
- PBX_E_INVOKE_FAILURE

Examples
This code invokes the DataWindow Update function and returns its integer return value:

```c
pbclass cls;
pbmethodID mid;
PBCallInfo* ci = new PBCallInfo;
pbint ret_val;

cls = session->GetClass(dwobj);
mid = session->GetMethodID
   (cls, "Update", PBRT_FUNCTION, "I");
session->InitCallInfo(cls, mid, ci);

session->InvokeObjectFunction(dwobj, mid, ci);

ret_val = ci.returnValue->GetInt();
session->FreeCallInfo(ci);
delete ci;
return ret_val;
```

See also
InvokeClassFunction
IsArrayItemNull
Description Returns true if the array item contains a null value; otherwise it returns false.
Syntax IsArrayItemNull(parray array, pblong dim[ ])

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid parray structure that you want to check for a null-valued array item.</td>
</tr>
<tr>
<td>dim</td>
<td>A pblong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of array.</td>
</tr>
</tbody>
</table>

Return value pbboolean.
See also GetArrayItemType, Set<type>ArrayItem, SetArrayItemToNull

IsAutoInstantiate
Description Returns true if the specified class is an autoinstantiated class; otherwise it returns false.
Syntax IsAutoInstantiate(pbclass)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle or structure</td>
</tr>
</tbody>
</table>

Return value pbboolean.

IsFieldArray
Description Returns true if the field of the specified object is an array; otherwise it returns false.
Syntax IsFieldArray(pbclass cls, pbfield fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle for the class whose field is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the specified object</td>
</tr>
</tbody>
</table>

Return value pbboolean.
This code tests whether the field identified by $fid$ is an array, and if so, gets the array value:

```c
fid = session->GetFieldID(cls, "arr_val");
if (session->IsFieldArray(cls, fid))
{
    arr_val = session->GetArrayField(myobj, fid, isNull);
    ...
}
```

See also
- GetFieldID
- GetFieldType
- Get<type>Field
- GetNumOfFields
- IsFieldNull
- IsFieldObject
- SetFieldToNull
- Set<type>Field

### IsFieldNull

**Description**

Returns true if the field of the specified object is a null value; otherwise it returns false.

**Syntax**

```c
IsFieldNull(pbobject obj, pbfield fid)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$obj$</td>
<td>A valid object handle for the object whose field is to be accessed</td>
</tr>
<tr>
<td>$fid$</td>
<td>The field ID of the specified object</td>
</tr>
</tbody>
</table>

**Return value**

`pbboolean`.

**Examples**

These statements test whether the field identified by $fid$ is null:

```c
fid = session -> GetFieldID(cls, "i_val");
if (session -> IsFieldNull(myobj, fid))
```

See also
- GetFieldID
- GetFieldType
- Get<type>Field
- GetNumOfFields
- IsFieldArray
- IsFieldObject
- SetFieldToNull
- Set<type>Field
**IsFieldObject**

**Description**
Returns true if the field of the specified object is an object; otherwise it returns false.

**Syntax**
`IsFieldObject(pbclass cls, pbfield fid)`

**Argument** | **Description**
--- | ---
`cls` | A valid class handle for the class whose field is to be accessed
`fid` | The field ID of the specified object

**Return value**
`pbboolean`

**Examples**
These statements test whether the field identified by `fid` is an object:
```c
fid = session -> GetFieldID(cls, "obj_val");
if (session -> IsFieldObject(myobj, fid))
```

**See also**
GetFieldID, GetFieldType, Get<type>Field, GetNumOfFields, IsFieldArray, IsFieldNull, SetFieldToNull, Set<type>Field

**IsGlobalVarArray**

**Description**
Returns true if the global variable contains an array; otherwise it returns false.

**Syntax**
`IsGlobalVarArray(pbfield fid)`

**Argument** | **Description**
--- | ---
`fid` | The field ID of the global variable

**Return value**
`pbboolean`

**Examples**
These statements test whether the field identified by `fid` is a global variable array:
```c
fid = session -> GetGlobalVarID("arr_gvar");
if (session -> IsGlobalVarArray(fid))
{
    arr_val=session -> GetArrayGlobalVar(fid, isNull);
    ...
```
IPB_Session interface

See also

- GetGlobalVarID
- GetGlobalVarType
- Get<type>GlobalVar
- IsGlobalVarNull
- IsGlobalVarObject
- SetGlobalVarToNull
- Set<type>GlobalVar

IsGlobalVarNull

Description

Returns true if the global variable contains a null value; otherwise it returns false.

Syntax

IsGlobalVarNull( pbfield fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fid</td>
<td>The field ID of the global variable</td>
</tr>
</tbody>
</table>

Return value

pbboolean.

Examples

These statements test whether the field identified by fid is a global variable array:

```plaintext
fid = session -> GetGlobalVarID("arr_gvar");
if (session -> IsGlobalVarArray(fid))
{
    arr_val=session -> GetArrayGlobalVar(fid, isNull);
    ...
```

See also

- GetGlobalVarID
- GetGlobalVarType
- Get<type>GlobalVar
- IsGlobalVarArray
- IsGlobalVarObject
- SetGlobalVarToNull
- Set<type>GlobalVar
IsGlobalVarObject

**Description**
Returns true if the global variable contains an object; otherwise it returns false.

**Syntax**
IsGlobalVarObject( pbfield *fid*)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>fid</em></td>
<td>The field ID of the global variable</td>
</tr>
</tbody>
</table>

**Return value**
{pbboolean}.

**Examples**
These statements test whether the field identified by *fid* is a global variable object. If it is, its value is set to another global variable object:

```c
    fid = session -> GetGlobalVarID("obj2_gvar");
    if (session -> IsGlobalVarObject(fid))
    {
        obj_val = session -> GetObjectGlobalVar(fid, isNull);
        cls = session -> GetClass(obj_val);
        fid = session -> GetFieldID(cls, "text");
        s_val = session -> GetStringField(obj_val, fid, isNull);
        mystr = session -> GetString(s_val);
        // Set the value of obj2_gvar to obj1_gvar
        fid = session -> GetGlobalVarID("obj1_gvar");
        session -> SetObjectGlobalVar(fid, obj_val);
    }
```

**See also**
GetGlobalVarID
GetGlobalVarType
Get<const GlobalVar>
IsGlobalVarArray
IsGlobalVarNull
SetGlobalVarToNull
Set<const GlobalVar>
IsNativeObject

Description Determines whether a pbobject is an instance of a native class.

Syntax IsNativeObject(pbobject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid object handle</td>
</tr>
</tbody>
</table>

Return value pbboolean.

Examples The f_getrow function uses IsNativeObject to test whether extObj is a native class. If so, it gets the native interface and invokes the f_getrowcount function in the other class:

```c
long f_getrow(IPB_Session* session, pbobject dwObj, pbobject extObj)
{
    long lRet;
    Imy_pbni* pImy_pbni = NULL;
    IPBX_NonVisualObject* pp=NULL;

    if (session -> IsNativeObject(extObj) )
    {
        pp = (IPBX_NonVisualObject*) session ->
            GetNativeInterface(extObj);
        pImy_pbni = static_cast<Imy_pbni*>(pp);
        lRet = pImy_pbni-> f_GetRowCount(session, dwObj);
    }
    return lRet;
}
```

Usage Use this method in conjunction with GetNativeInterface to obtain a direct reference to the IPBX_UserObject associated with another native class, so that the class and its methods can be accessed directly.

See also GetNativeInterface
**IsSharedVarArray**

**Description**
Returns true if the shared variable contains an array; otherwise it returns false.

**Syntax**
IsSharedVarArray(pbgroup *group, pbfield *fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group whose shared variable is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the shared variable</td>
</tr>
</tbody>
</table>

**Return value**
pbboolean.

**See also**
Get-type>SharedVar
GetSharedVarID
GetSharedVarType
IsSharedVarNull
IsSharedVarObject
Set-type>SharedVar
SetSharedVarToNull

**IsSharedVarNull**

**Description**
Returns true if the shared variable contains a null value; otherwise it returns false.

**Syntax**
IsSharedVarNull(pbgroup *group, pbfield *fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group whose shared variable is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the shared variable</td>
</tr>
</tbody>
</table>

**Return value**
pbboolean.

**See also**
Get-type>SharedVar
GetSharedVarID
GetSharedVarType
IsSharedVarArray
IsSharedVarObject
Set-type>SharedVar
SetSharedVarToNull
### `IsSharedVarObject`

**Description**
Returns true if the shared variable contains an object; otherwise it returns false.

**Syntax**
```
IsSharedVarObject(pbgroup group, pbfield fid)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group whose shared variable is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the shared variable</td>
</tr>
</tbody>
</table>

**Return value**
pbboolean.

**See also**
Get<type>SharedVar
GetSharedVarID
GetSharedVarType
IsSharedVarArray
IsSharedVarNull
Set<type>SharedVar
SetSharedVarToNull

### `NewBlob`

**Description**
Creates a new blob and duplicates a buffer for the new blob data.

**Syntax**
```
NewBlob (const void* bin, pblong len)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>A void pointer that points to the source buffer</td>
</tr>
<tr>
<td>len</td>
<td>The length in bytes of the data in the buffer</td>
</tr>
</tbody>
</table>

**Return value**
pbblob.

**Examples**
If the blob value in the PBCallInfo structure is null, this code creates a new blob value with four bytes in the `pArguments` array; otherwise, it sets the blob value in the `pArguments` array to the value in the PBCallInfo structure:
```
if (ci->pArgs->GetAt(i)->IsNull())
    pArguments[i].blob_val =
        Session->NewBlob("null", 4);
else
    pArguments[i].blob_val =
        ci->pArgs->GetAt(i)->GetBlob();
```

**Usage**
The buffer containing the new blob data is freed when `PopLocalFrame` is called.

**See also**
PopLocalFrame
SetBlob
**NewBoundedObjectArray**

Description: Creates a bounded PowerBuilder object or structure array.

Syntax:  
```c
NewBoundedObjectArray(pbclass cls, pbuint dimension,  
                      PBArrayInfo::ArrayBound* bounds)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle of the type of PowerBuilder object or structure array to be created</td>
</tr>
<tr>
<td>dimension</td>
<td>A number greater than one that indicates the dimension of the array to be created</td>
</tr>
<tr>
<td>bounds</td>
<td>An array containing the upper and lower boundaries of the array to be created</td>
</tr>
</tbody>
</table>

Return value: pbarray or null on failure.

Examples:
```c
int size;
pbarray pbin_a;
PBAArrayInfo* ai;
PBXRESULT ret;
pbclass cls;
pbgroup group;

size = sizeof(PBArrayInfo) +  
      sizeof(PBArrayInfo::ArrayBound);
ai = (PBAArrayInfo*)malloc(size);
ai-> bounds[0].upperBound=2;
ai-> bounds[0].lowerBound=1;
ai-> bounds[1].upperBound=2;
ai-> bounds[1].lowerBound=1;
ai-> numDimensions=2;

// Create new array pbin_a
group = session->FindGroup("w_main", pbgroup_window);
if (group==NULL)  
  return;
cls = session->FindClass(group, "commandbutton");
if( cls==NULL)  
  return;
pbin_a = session->NewBoundedObjectArray(cls,  
                                          ai-> numDimensions, ai-> bounds);
```

See also:  
Get<type>ArrayItem  
GetArrayInfo  
GetArrayItemType  
GetArrayLength
IPB_Session interface

IsArrayItemNull
NewBoundedSimpleArray
NewUnboundedObjectArray
NewUnboundedSimpleArray
ReleaseArrayInfo
Set<type>ArrayItem
SetArrayItemToNull
SetArrayItemValue

NewBoundedSimpleArray

Description
Creates a bounded simple data array.

Syntax
NewBoundedSimpleArray(pbuint type, pbuint dimension,
PBArrrayInfo::ArrayBound* bounds)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>An enumerated variable of type pbvalue_*, indicating the type of simple unbounded array to be created</td>
</tr>
<tr>
<td>dimension</td>
<td>A number greater than one that indicates the dimension of the array to be created</td>
</tr>
<tr>
<td>bounds</td>
<td>An array containing the upper and lower boundaries of the array to be created</td>
</tr>
</tbody>
</table>

Return value
pbarray or null on failure.

See also
Get<type>ArrayItem
GetArrayInfo
GetArrayItemType
GetArrayLength
IsArrayItemNull
NewBoundedObjectArray
NewUnboundedObjectArray
NewUnboundedSimpleArray
ReleaseArrayInfo
Set<type>ArrayItem
SetArrayItemToNull
SetArrayItemValue
NewDate

Description
Creates a new pbdate data object.

Syntax
NewDate()

Return value
pbdate.

Examples
This example tests whether a date value exists, and, if it does not, it creates a new pbdate object and sets its value to the first day in January, 1900:

```cpp
if (ci->pArgs->GetAt(0)->IsNull())
{
    pArguments[i].date_val = Session->NewDate();
    Session->SetDate(pArguments[i].date_val, 1900,1,1); // Date: 1900-01-01
    isNull[i]=true;
}
else
{
    pArguments[i].date_val = ci->pArgs->GetAt(i)->GetDate();
    isNull[i]=false;
}
```

Usage
The initial value is 1900-1-1.

See also
SetDate
SplitDate
**NewDateTime**

Description: Creates a new pbdatetime data object.

Syntax: `NewDateTime()`

Return value: `pbdatetime`.

Examples: This example tests whether a date/time value exists, and, if it does not, it creates a new pbdate object and sets its value to the beginning of January, 1900:

```c
if (ci->pArgs->GetAt(i)->IsNull())
{
    pArguments[i].datetime_val=Session->NewDateTime();
    Session->SetDateTime(pArguments[i].datetime_val,
                1900, 1, 1, 1, 1, 1); // Datetime:
                // 1900-01-01 01:01:01
}
else
{
    pArguments[i].datetime_val =
        ci->pArgs->GetAt(i)->GetDateTime();
}
```

Usage: The initial value is 1900-1-1 0:0:0.0.

See also: `SetDateTime`, `SplitDateTime`.

**NewDecimal**

Description: Allocates resources for a new decimal data object.

Syntax: `NewDecimal()`

Return value: `pbdec` or `null` on failure.

Examples: This example tests whether a date/time value exists, and, if it does not, it creates a new pbdate object and sets its value to the beginning of January, 1900:

```c
if (ci->pArgs->GetAt(i)->IsNull())
{
    pArguments[i].dec_val=Session->NewDecimal();
    Session->SetDecimal(pArguments[i].dec_val,"1.0");
}
else
{
    pArguments[i].dec_val =
        ci->pArgs->GetAt(i)->GetDecimal();
}
```

See also: `GetDecimalString`, `ReleaseDecimalString`, `SetDecimal`.

---

**IPB_Session interface**

---

164 PowerBuilder Classic
NewObject

Description: Creates a new object of the specified type.

Syntax: `NewObject(pbclass cls)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cls</code></td>
<td>The type of object or structure instance to be created</td>
</tr>
</tbody>
</table>

Return value: `pbobject` of the given class or structure.

Examples:
```
pbclass cls;
pbobject ex;
pbgroup group;

    group = session->FindGroup("user_exception", pbgroup_userobject);
    if (group==NULL)
        return;
    cls = session->FindClass(group, "user_exception");
    if (group==NULL)
        return;
    ex = session->NewObject(cls);
```

Usage: The returned object’s life cycle is restricted to the current frame unless `AddGlobalRef` is called on the object.

See also: FindClass, FindGroup
**IPB_Session interface**

### NewProxyObject

**Description**
Creates a proxy for a remote object. The proxy is used to extend the network protocol in PowerBuilder.

**Syntax**
```c
NewProxyObject(pbclass cls)
```

**Return value**
pbproxyobject.

**Examples**
This example creates a new proxy object, creates a marshaler, and associates the marshaler with the proxy object:

```c
pbproxyObject proxy = session->NewProxyObject(cls);
if (proxy == NULL) {
    ci->returnValue->SetLong(kFailToCreateProxy);
    return PBX_OK;
}

// Create MyMarshaler
MyMarshaler* marshaler = new MyMarshaler(env, proxy, obj);

// Associate MyMarshaler with the proxy
session->SetMarshaler(proxy, marshaler);

ci->pArgs->GetAt(0)->SetObject(proxy);

ci->returnValue->SetLong(kSuccessful);
return PBX_OK;
```

**See also**
GetMarshaler
SetMarshaler
NewString

Description Creates a new string.
Syntax NewString(LPCTSTR)
Return value pbstring.
Examples

```c
pbclass cls;
cls = session->GetSystemFunctionsClass();
if( cls == NULL )
{
    ret_val = session->NewString("null");
    return ret_val;
}
```
Usage The returned string is destroyed when PopLocalFrame is called.
See also SetString

NewTime

Description Creates a new pbtime data object.
Syntax NewTime()
Return value pbtime.
Examples These statements split a time into hours, minutes, and seconds, and then use the resulting values to set the value of a new time object:

```c
Session->SplitTime(ci.returnValue->GetTime(), &hh, &mm, &ss);
ret_val = Session->NewTime();
Session->SetTime(ret_val, hh, mm, ss);
```
Usage The initial value is 0:0:0.0.
See also SetTime
   SplitTime
NewUnboundedObjectArray

Description
Creates an unbounded PowerBuilder object or structure data array.

Syntax
NewUnboundedObjectArray(pbclass cls)

Argument | Description
---------|---------------------------
cls       | A valid class handle of the type of PowerBuilder object or structure array to be created

Return value
pbarray or null on failure.

Usage
An unbounded array can have only one dimension, so no dimension information is needed.

See also
Get<type>ArrayItem
GetArrayInfo
GetArrayItemType
GetArrayLength
IsArrayItemNull
NewBoundedObjectArray
NewBoundedSimpleArray
NewUnboundedSimpleArray
ReleaseArrayInfo
Set<type>ArrayItem
SetArrayItemToNull
SetArrayItemValue
**NewUnboundedSimpleArray**

**Description**
Creates an unbounded simple data array.

**Syntax**
```
NewUnboundedSimpleArray(pbuint type)
```

**Argument**
```
type
```
An enumerated variable of type `pbvalue_*` indicating the type of simple unbounded array to be created.

**Return value**
pbarray or null on failure.

**Examples**
This example creates an unbounded simple data array of the type returned by the `getDataType` method, which returns a string of the form `dt_type`. Most of the `case` statements have been removed for the sake of brevity:

```java
if (d_returnType.isArray())
{
  returnValue.l = env->CallObjectMethodA(obj,
    mid, values.get());
  pbarray v;

  switch(d_returnType.getDataType())
  {
    case dt_boolean:
      v = session->NewUnboundedSimpleArray
        (pbvalue_boolean);
      break;

    case dt_short:
      v = session->NewUnboundedSimpleArray
        (pbvalue_int);
      break;

    // CASE statements omitted
    ...

    default:
      v = session->NewUnboundedSimpleArray
        (pbvalue_any);
      break;
  }

  ci->returnValue->SetArray(v);
}
```

**Usage**
An unbounded array can have only one dimension, so no dimension information is needed.
See also
Get<type>ArrayItem
GetArrayInfo
GetArrayItemType
GetArrayLength
IsArrayItemNull
NewBoundedObjectArray
NewBoundedSimpleArray
NewUnboundedObjectArray
ReleaseArrayInfo
Set<type>ArrayItem
SetArrayItemToNull
SetArrayItemValue

PopLocalFrame
Description
Pops the current local reference frame from the current native method stack
frame, removing all local references to the objects added in that local frame.
All the pbobject, pbstring, and pbdecimal variables created by calling
NewDecimal, NewObject, or NewString in the current frame are destroyed
automatically.

Syntax
PopLocalFrame()

Return value
None.

See also
AddLocalRef
PushLocalFrame
RemoveLocalRef

ProcessPBMessage
Description
Checks the PowerBuilder message queue and, if there is a message in the
queue, attempts to process it.

Syntax
ProcessPBMessage()

Return value
pbboolean. Returns true if a PowerBuilder message was processed, and false
otherwise.
Examples

This message loop in a WinMain function processes a PowerBuilder message if a message has been received and an IPB session is running:

```c
try {
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);

        // Call to ProcessPBMessage
        if (session)
            session->ProcessPBMessage();
    }
}
```

This overloaded WindowProc function in an MFC application processes a PowerBuilder message:

```c
LRESULT CCallPBVCtrl::WindowProc(UINT message, WPARAM wParam, LPARAM lParam) {
    d_session->ProcessPBMessage();
    return CDialog::WindowProc(message, wParam, lParam);
}
```

Usage

Each time this function is called, it attempts to retrieve a message from the PowerBuilder message queue and process it. It is similar to the PowerBuilder Yield function; however, ProcessPBMessage processes only one message at a time, and it processes only PowerBuilder messages. The Yield function also processes Windows messages.

Use this function when PowerBuilder windows or visual controls are called from C++ applications or from extensions to ensure that events posted to the PowerBuilder message queue are processed.

If the function is not inserted in the C++ application in a way that results in it being called repeatedly, posted events are not processed in the PowerBuilder application.

For most applications, ProcessPBMessage can be inserted in a message loop in the WinMain function. If you use Microsoft Foundation Classes (MFC), you cannot modify the built-in message loop. To ensure that the ProcessPBMessage function is called repeatedly, you can overload the CWnd::WindowProc function and insert ProcessPBMessage into the overloaded function.
**IPB_Session interface**

### PushLocalFrame

description: Pushes a local reference frame onto the current native method stack frame. A local frame is analogous to a scope in C++.

**Syntax**
PushLocalFrame()

**Return value**
None.

**See also**
PopLocalFrame
RemoveLocalRef

### Release

description: Releases the current IPB_Session. The IPB_Session object becomes invalid after the call.

**Syntax**
Release()

**Return value**
None.

**Examples**
This example shows a call to Release. The example checks whether there is a valid session object before attempting to release it:

```c
if (pIPB_ObjectFactory)
{
    pIPB_ObjectFactory->Release();
    pIPB_ObjectFactory = NULL;
}
```

### ReleaseArrayInfo

description: Releases memory returned by GetArrayInfo.

**Syntax**
ReleaseArrayInfo(PBArrayInfo* pbarrayinfo)

**Argument | Description**

| pbarrayinfo | A valid PBArrayInfo handle |

**Return value**
PBXRESULT. PBX_OK for success.

**Examples**
This example shows how ReleaseArrayInfo should be called when memory allocated by GetArrayInfo is no longer needed:

```c
PBArrayInfo* ai;
...
session->ReleaseArrayInfo(ai);
```
Usage
If the array is an unbounded array, the bounds information in PBArrainfo is undetermined.

See also
Get<type>ArrayItem
GetArrayInfo
GetArrayItemType
GetArrayLength
IsArrayItemNull
NewBoundedObjectArray
NewBoundedSimpleArray
NewUnboundedObjectArray
NewUnboundedSimpleArray
Set<type>ArrayItem
SetArrayItemToNull
SetArrayItemValue

ReleaseDateString
Description
Frees the memory acquired using GetDateString.
Syntax
ReleaseDateString(LPCTSTR string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string to be released from memory</td>
</tr>
</tbody>
</table>

Return value
None.
See also
GetDateString

ReleaseDateTimeString
Description
Frees the memory acquired using GetDateTimeString.
Syntax
ReleaseDateTimeString(LPCTSTR string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string to be released from memory</td>
</tr>
</tbody>
</table>

Return value
None.
See also
GetDateTimeString
**ReleaseDecimalString**

**Description**
Frees the memory acquired using GetDecimalString.

**Syntax**
ReleaseDecimalString(LPCTSTR string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string to be released from memory</td>
</tr>
</tbody>
</table>

**Return value**
None.

**See also**
GetDecimalString

**ReleaseResultSetAccessor**

**Description**
Releases the pointer obtained using GetResultSetAccessor.

**Syntax**
ReleaseResultSetAccessor (IPB_ResultSetAccessor* rs)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>A pointer to the IPB_ResultSetAccessor object to be released</td>
</tr>
</tbody>
</table>

**Return value**
None.

**Examples**
This statement releases the IPB_ResultSetAccessor object rsa:

```
Session->ReleaseResultSetAccessor(rsa);
```

**Usage**
When you call ReleaseResultSetAccessor, the Release function of the IPB_ResultSetAccessor interface is called on the rs argument to release the interface pointer.

**See also**
CreateResultSet
GetResultSetAccessor
ReleaseString

Description
Frees the memory acquired using GetString, GetClassName, GetFieldName, or GetEnumItemName.

Syntax
ReleaseString(LPCTSTR string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string to be released from memory</td>
</tr>
</tbody>
</table>

Return value
None.

Examples
The following example gets a pointer to each of two strings passed in as arguments, concatenates them in a new string, then releases the memory used by the original strings:

```cpp
pbstring psppcls:: f_add_string(IPB_Session* session, pbstring arg1, pbstring arg2)
{
    LPCTSTR pStr1,pStr2;
    TCHAR tmp[100];
    pbstring ret;

    pStr1=session->GetString(arg1);
    pStr2=session->GetString(arg2);
    _tcscpy(tmp,pStr1);
    _tcscat(tmp,pStr2);
    ret = session -> NewString(tmp);
    session->ReleaseString(pStr1);
    session->ReleaseString(pStr2);

    return ret;
}
```

Usage
Do not use this function to release a string obtained using GetDateString, GetTimeString, GetDateTimeString, or GetDecimalString. Each of these Get methods has a corresponding Release method.

See also
GetClassName
GetEnumItemName
GetFieldName
GetString
**IPB_Session interface**

**ReleaseTimeString**

Description Frees the memory acquired using GetTimeString.

Syntax `ReleaseTimeString(LPCTSTR string)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>The string to be released from memory</td>
</tr>
</tbody>
</table>

Return value None.

See also GetTimeString

**ReleaseValue**

Description Frees the IPB_Value acquired using AcquireValue or AcquireArrayItemValue.

Syntax `ReleaseValue(IPB_Value* value)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The string to be released from memory</td>
</tr>
</tbody>
</table>

Return value None.

Examples The AcquireValue method is used to obtain a message argument value. Later, when the value is no longer needed, it is released using ReleaseValue to avoid memory leaks:

```cpp
// Acquire a value
MessageArg = session->AcquireValue
  ( ci->pArgs->GetAt(0) );
pbstring pbMessage = MessageArg->GetString() ;
Message = (LPSTR)session->GetString(pbMessage) ;
...
// Cleanup phase
if (MessageArg) {
  Session->ReleaseValue ( MessageArg ) ;
}
```
Usage

When you no longer need the data acquired using the AcquireValue or AcquireArrayItemValue method, you must call the ReleaseValue method to free the data. Failing to do so causes a memory leak.

**Warning!** Do not use ReleaseValue to release a value that was not acquired using AcquireValue or AcquireArrayItemValue. If you do, the PowerBuilder VM might crash.

See also

AcquireArrayItemValue
AcquireValue

RemoveGlobalRef

Description

Removes a global reference to the specified PowerBuilder object.

Syntax

RemoveGlobalRef (pbobject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid PowerBuilder object handle</td>
</tr>
</tbody>
</table>

Return value

None.

Examples

```c
void MyPBNIClass::reference()
{
    d_session->AddGlobalRef(d_pbobject);
}
```

```c
void MyPBNIClass::unreference()
{
    if(d_pbobject != NULL)
        d_session -> RemoveGlobalRef(d_pbobject);
}
```

See also

AddGlobalRef
**RemoveLocalRef**

**Description**
Removes a local reference to the specified PowerBuilder object.

**Syntax**
RemoveLocalRef (pobject  *obj*)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>obj</em></td>
<td>A valid PowerBuilder object handle</td>
</tr>
</tbody>
</table>

**Return value**
None.

**See also**
AddLocalRef  
PopLocalFrame  
PushLocalFrame

**RemoveProp**

**Description**
Removes the specified variable from the list of properties of the current IPB session. You must free the memory to which the property points.

**Syntax**
RemoveProp(LPCTSTR  name)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the variable to be removed</td>
</tr>
</tbody>
</table>

**Return value**
None.

**Examples**
These statements remove prop_name from the list of variables associated with the session and delete the pointer created to point to the variables value:

```c
    session -> RemoveProp(prop_name);
    delete SetValue;
```

**Usage**
*SetProp* enables you to use a variable value throughout an IPB session. Use *RemoveProp* to remove the variable from the list of variables associated with the session when it is no longer needed. You must also free the memory associated with the variable.

**See also**
GetProp  
SetProp
**RestartRequested**

**Description**
Determines whether the PowerBuilder system function Restart has been called.

**Syntax**
HasPBVisualObject()

**Return value**
pbboolean. Returns true when the PowerBuilder system function Restart is called. When RestartRequested returns true, you should destroy the existing IPB_Session object and create a new one to restart the application.

**Examples**
In the following example, StartApplication, RecreateSession, and CleanApplication are functions of the PB_MyConsoleAppRunner class. StartApplication is similar to the IP_VM RunApplication function, but it uses an existing session. RecreateSession releases the current session and creates a new one. CleanApplication triggers the application’s Close event and releases resources. In the example, RestartRequested is called in a DO loop to test whether the PowerBuilder Restart function has been called. If it has, the RecreateSession function is called:

```cpp
PBXRESULT PB_MyConsoleAppRunner::RunApplication()
{
    PBXRESULT res;
    pbboolean restart = FALSE;
    do
    {
        res = StartApplication();
        if (res != PBX_OK)
            break;
        restart = GetSession()->RestartRequested();
        if (restart)
            RecreateSession();
    } while (restart);
    return CleanApplication();
}
```

**Usage**
RestartRequested and HasVisualPBObject are used in the implementation of the IPB_VM RunApplication function. You no longer need to use an external message loop to check for Windows messages when you call the RunApplication function as you did in versions of PBN1 prior to PowerBuilder 10.5.

**See also**
HasPBVisualObject
RunApplication
**Set<type>ArrayItem**

**Description**
Assigns a value to an array item of a specific type.

**Syntax**

```
SetBlobArrayItem ( pbarray array, pblong dim[ ], ppblob value )
SetBoolArrayItem ( pbarray array, pblong dim[ ], ppboolean value )
SetByteArrayItem ( pbarray array, pblong dim[ ], pbbyte value )
SetCharArrayItem ( pbarray array, pblong dim[ ], pbchar value )
SetDateArrayItem ( pbarray array, pblong dim[ ], pbdate value )
SetDateTimeArrayItem ( pbarray array, pblong dim[ ], pbdatetime value )
SetDecArrayItem ( pbarray array, pblong dim[ ], pbdec value )
SetDoubleArrayItem ( pbarray array, pblong dim[ ], pbdouble value )
SetIntArrayItem ( pbarray array, pblong dim[ ], pbint value )
SetLongArrayItem ( pbarray array, pblong dim[ ], pblong value )
SetLongLongArrayItem ( pbarray array, plonglong dim[ ], pblong value )
SetObjectArrayItem ( pbarray array, pblong dim[ ], pbobject obj )
SetPBStringArrayItem ( pbarray array, pblong dim[ ], pbstring value )
SetRealArrayItem ( pbarray array, pblong dim[ ], pbreal value )
SetStringArrayItem ( pbarray array, pblong dim[ ], LPCTSTR value )
SetTimeArrayItem ( pbarray array, pblong dim[ ], pbtime value )
SetUintArrayItem ( pbarray array, pblong dim[ ], pbuint value )
SetUlongArrayItem ( pbarray array, pblong dim[ ], pbulong value )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid pbarray handle.</td>
</tr>
<tr>
<td>dim</td>
<td>A pblong array to hold indexes of each dimension. The number of dimensions must equal the number of dimensions of the array.</td>
</tr>
<tr>
<td>value</td>
<td>The new value of the array item.</td>
</tr>
</tbody>
</table>

**Return value**
PBXRESULT. PBX_OK for success.

If the index exceeds the bounds of a bounded array, it returns PBX_E_ARRAY_INDEX_OUTOF_BOUNDS.

If the data passed in does not match the datatype of the array, it returns PBX_E_MISMATCHED_DATA_TYPE.

**Examples**
This example creates a new unbounded simple array. In the FOR loop, application-specific code (not shown here) gets array values, which are then added to the array using SetPBStringArrayItem:
pblong dim[1];
char * cstr;
pbuint numDimensions = 1;
PBArrrayInfo::ArrayBound bound;

bound.lowerBound = 1;
bound.upperBound = size;
d_pbarray = d_session->NewBoundedSimpleArray
    (pbvalue_string, numDimensions, &bound);

for (int i = 1; i <= size; i++)
{
    dim[0] = i;
    // add application-specific code here to
    // get array value
    pbstring pValue = d_session->NewString(cstr);
    d_session->SetPBStringArrayItem(d_pbarray, dim, 
        pValue);

    delete [] cstr;
}
 pbv.SetArray(d_pbarray);

Usage
This method assigns the IPB_Value pointed to by the value argument to the
array item in the same way that the IPB_Value Set<type> method sets a value.

See also
Get<type>ArrayItem
GetArrayInfo
GetArrayItemType
GetArrayLength
IsArrayItemNull
NewBoundedObjectArray
NewBoundedSimpleArray
NewUnboundedObjectArray
NewUnboundedSimpleArray
ReleaseArrayInfo
SetArrayItemToNull
SetArrayItemTypeValue
**Set<type>Field**

**Description**
A set of methods that set a new value in an instance field of an object.

**Syntax**
- `SetArrayField ( pbobject obj, pbfieldID fid, parray value )`
- `SetBlobField ( pbobject obj, pbfieldID fid, ppblob value )`
- `SetBoolField ( pbobject obj, pbfieldID fid, ppboolean value )`
- `SetByteField ( pbobject obj, pbfieldID fid, ppbyte value )`
- `SetCharField ( pbobject obj, pbfieldID fid, ppchar value )`
- `SetDateField ( pbobject obj, pbfieldID fid, ppdate value )`
- `SetDateTimeField ( pbobject obj, pbfieldID fid, ppdatetime value )`
- `SetDecField ( pbobject obj, pbfieldID fid, ppdec value )`
- `SetDoubleField ( pbobject obj, pbfieldID fid, ppdouble value )`
- `SetIntField ( pbobject obj, pbfieldID fid, ppint value )`
- `SetLongField ( pbobject obj, pbfieldID fid, pplong value )`
- `SetLongLongField ( pbobject obj, pbfieldID fid, pplonglong value )`
- `SetObjectField ( pbobject obj, pbfieldID fid, ppobject value )`
- `SetPBStringField ( pbobject obj, pbfieldID fid, ppstring value )`
- `SetRealField ( pbobject obj, pbfieldID fid, ppreal value )`
- `SetRealField ( pbobject obj, pbfieldID fid, pctype value )`
- `SetTimeStringField ( pbobject obj, pbfieldID fid, ppstring value )`
- `SetTimeField ( pbobject obj, pbfieldID fid, pptime value )`
- `SetUintField ( pbobject obj, pbfieldID fid, ppuuint value )`
- `SetUlongField ( pbobject obj, pbfieldID fid, ppulong value )`

**Return value**
PBX_RESULT.

**Examples**
These statements set a new string value in a string field:

```c
pbstring str = session->NewString(d_message.c_str());
if (str != NULL)
    session->SetPBStringField(d_pbobj, d_fidMsg, str);
```
Usage

When you change any visual property of a PowerBuilder object by calling `Set<type>field` functions, the property is changed but the property is not refreshed in the graphical user interface. `UpdateField` refreshes the visual properties of PowerBuilder objects. You must call `UpdateField` explicitly when changing any visual property with the `Set<type>field` functions.

See also

- `GetFieldID`
- `GetFieldType`
- `Get<type>Field`
- `GetNumOfFields`
- `IsFieldArray`
- `IsFieldNull`
- `IsFieldObject`
- `SetFieldToNull`
- `UpdateField`
Set\texttt{<type>\texttt{GlobalVar}}

Description  
A set of methods that set the value of a global variable of a specific datatype.

Syntax  
- \texttt{SetArrayGlobalVar ( pbfieldID \texttt{fid}, pbarray \texttt{value} )}
- \texttt{SetBlobGlobalVar ( pbfieldID \texttt{fid}, pblob \texttt{value} )}
- \texttt{SetBoolGlobalVar ( pbfieldID \texttt{fid}, pbboolean \texttt{value} )}
- \texttt{SetByteGlobalVar ( pbfieldID \texttt{fid}, pbbyte \texttt{value} )}
- \texttt{SetCharGlobalVar ( pbfieldID \texttt{fid}, pbchar \texttt{value} )}
- \texttt{SetDateGlobalVar ( pbfieldID \texttt{fid}, pbdate \texttt{value} )}
- \texttt{SetDateTimeGlobalVar ( pbfieldID \texttt{fid}, pbdatetime \texttt{value} )}
- \texttt{SetDecGlobalVar ( pbfieldID \texttt{fid}, pbdec \texttt{value} )}
- \texttt{SetDoubleGlobalVar ( pbfieldID \texttt{fid}, pbdouble \texttt{value} )}
- \texttt{SetIntGlobalVar ( pbfieldID \texttt{fid}, pbint \texttt{value} )}
- \texttt{SetLongGlobalVar ( pbfieldID \texttt{fid}, pblong \texttt{value} )}
- \texttt{SetLongLongGlobalVar ( pbfieldID \texttt{fid}, pblonglong \texttt{value} )}
- \texttt{SetObjectGlobalVar ( pbfieldID \texttt{fid}, pbobject \texttt{value} )}
- \texttt{SetPBStringGlobalVar ( pbfieldID \texttt{fid}, pbstring \texttt{value} )}
- \texttt{SetRealGlobalVar ( pbfieldID \texttt{fid}, pbreal \texttt{value} )}
- \texttt{SetStringGlobalVar ( pbfieldID \texttt{fid}, LPCTSTR \texttt{value} )}
- \texttt{SetTimeGlobalVar ( pbfieldID \texttt{fid}, pbtime \texttt{value} )}
- \texttt{SetUintGlobalVar ( pbfieldID \texttt{fid}, pbuint \texttt{value} )}
- \texttt{SetUlongGlobalVar ( pbfieldID \texttt{fid}, pbulong \texttt{value} )}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{fid}</td>
<td>The field ID of the global variable</td>
</tr>
<tr>
<td>\texttt{value}</td>
<td>The value to be set</td>
</tr>
</tbody>
</table>

Return value  
\texttt{PBX\_RESULT}.

Examples  
This shows how to add 1 to the value of a global variable:

\begin{verbatim}
    fid = session -> GetGlobalVarID("l_gvar");
    l_val = session -> GetLongGlobalVar(fid, isNull);
    session -> SetLongGlobalVar(fid, l_val + 1);
\end{verbatim}

See also  
- GetGlobalVarID
- GetGlobalVarType
- Get\texttt{<type>\texttt{GlobalVar}}
- IsGlobalVarObject
- SetGlobalVarToNull
Set\text少\textless type\textgreater\textgreater\textendash SharedVar

Description
A set of methods that set the value of a shared variable of a specific datatype.

Syntax
SetArraySharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, parray \textit{value} )

SetBlobSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbblob \textit{value} )

SetBoolSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbboolean \textit{value} )

SetByteSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbbyte \textit{value} )

SetCharSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbchar \textit{value} )

SetDateSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbdate \textit{value} )

SetDateTimeSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbdatetime \textit{value} )

SetDecSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbdec \textit{value} )

SetDoubleSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbdouble \textit{value} )

SetIntSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbint \textit{value} )

SetLongSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pblong \textit{value} )

SetLongLongSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pblonlong \textit{value} )

SetObjectSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbobject \textit{value} )

SetPBStringSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbstring \textit{value} )

SetRealSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbreal \textit{value} )

SetStringSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, LPCTSTR \textit{value} )

SetTimeSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbtime \textit{value} )

SetUintSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbuint \textit{value} )

SetUlongSharedVar ( pbgroup \textit{group}, pbfieldID \textit{fid}, pbulong \textit{value} )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{group}</td>
<td>The group whose shared variable is to be accessed</td>
</tr>
<tr>
<td>\textit{fid}</td>
<td>The field ID of the shared variable</td>
</tr>
<tr>
<td>\textit{value}</td>
<td>The value to be set</td>
</tr>
</tbody>
</table>

Return value
PBX\_RESULT.

See also
Get\textless type\textgreater\textless SharedVar
GetSharedVarID
GetSharedVarType
IsSharedVarArray
IsSharedVarNull
IsSharedVarObject
SetSharedVarToNull
IPB_Session interface

SetArrayItemToNull
Description
Sets the value of an array item to a null value.

Syntax
SetArrayItemToNull( pbarray array, pblong dim[] )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid pbarray structure in which you want to set an array item to null.</td>
</tr>
<tr>
<td>dim</td>
<td>A pblong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of array.</td>
</tr>
</tbody>
</table>

Return value  pbboolean.
See also  IsArrayItemNull

SetArrayItemValue
Description
Sets the value of an array item to the value of an IPB_Value.

Syntax
SetArrayItemValue( pbarray array, pblong dim[], IPB_Value* src )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid pbarray structure in which you want to set an array item to null.</td>
</tr>
<tr>
<td>dim</td>
<td>A pblong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of array.</td>
</tr>
<tr>
<td>src</td>
<td>The value to which the array item is to be changed.</td>
</tr>
</tbody>
</table>

Return value  None.
Examples
This code sets the value of each item in an array:

```c
for( i=1; i <= bound; i++ )
{
    dim[0]= i;
    ipv = Session -> AcquireArrayItemValue(refArg, dim);
    Session -> SetArrayItemValue(*i_array, dim, ipv);
    Session -> ReleaseValue(ipv);
}
```

Usage
The SetArrayItemValue method does not verify that the datatype of the replacement value matches the datatype of the original value.
SetBlob

Description
Destroys the existing data in a blob and copies data into it from a buffer.

Syntax
SetBlob (pbblob blb, const void* bin, pblong len)

Return value
PBXRESULT. Returns PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new blob value is invalid; otherwise, returns PBX_E_OUTOF_MEMORY.

Usage
A deep copy is performed. The existing value is destroyed first, and then the contents of the bin argument are copied into a new value.

See also
NewBlob

SetDate

Description
Resets the value of the specified pbdate object.

Syntax
SetDate (pbdate date, pbint year, pbint month, pbint day)

Return value
PBXRESULT. PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new date is invalid.

Examples
This example sets the date to March 12, 1938:

session->SetDate(date_val, 1938, 3, 12);
IPB_Session interface

Usage
If the parameters are invalid, the date is reset to 1900-1-1.

See also
NewDate
SplitDate

SetDateTime

Description
Resets the value of the specified pbdatetime object.

Syntax
SetDate (pbdatetime dt, pbint year, pbint month, pbint day, pbint hour, pbint minute, pbdouble second)

Argument | Description
----------|-----------------------------
dt | The pbdatetime object to be reset
year | A year in the range 1000 to 3000
month | A month in the range 1 to 12
day | A day in the range 1 to 31
hour | An hour in the range 0 to 23
minute | A minute in the range 0 to 59
second | A second in the range 0 to 59.999999

Return value
PBX_RESULT. PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new datetime is invalid.

Examples
This example sets the datetime value to August 19, 1982 at 10:30:45.10:

```cpp
session->SetDate(date_val, 1982, 8, 19, 10, 30, 45.1);
```

Usage
If the parameters are invalid, the datetime value is reset to 1900-1-1 0:0:0.0.

See also
NewDateTime
SplitDateTime
SetDecimal

Sets the value of a decimal variable to decimal data in a string.

Syntax

SetDecimal(pbdec dec, LPCTSTR dec_str)

Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec</td>
<td>The decimal data object to be set</td>
</tr>
<tr>
<td>dec_str</td>
<td>The string containing the data to be converted to a decimal</td>
</tr>
</tbody>
</table>

Return value

PBXRESULT. PBX_OK for success.

Examples

This example uses the IPB_Session SetDecimal method to set the value of a variable of type pbdec, then uses the IPB_Value SetDecimal method to set the return value in the PBCallInfo structure:

```c
pbdec pbdecRet = NULL;
LPTSTR lpDecValueToReturn = NULL;
...
pbdecRet = session -> NewDecimal();
session -> SetDecimal( pbdecRet,
    (LPCTSTR)lpDecValueToReturn);
ci -> returnValue -> SetDecimal(pbdecRet);
```

Usage

If the string contains invalid data, the decimal value is set to 0.0.

See also

GetDecimalString
NewDecimal
ReleaseDecimalString
### SetFieldToNull

**Description**
Sets the value of the specified field to null.

**Syntax**
```
SetFieldToNull(pbobject obj, pbfield fid)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj</code></td>
<td>A valid object handle</td>
</tr>
<tr>
<td><code>fid</code></td>
<td>The field ID of the specified object</td>
</tr>
</tbody>
</table>

**Return value**
None.

**See also**
- GetFieldID
- GetFieldType
- Get<type>Field
- GetNumOfFields
- IsFieldArray
- IsFieldNull
- IsFieldObject
- Set<type>Field

### SetGlobalVarToNull

**Description**
Sets the value of the specified global variable to null.

**Syntax**
```
SetGlobalVarToNull(pbobject obj, pbfield fid)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fid</code></td>
<td>The field ID of the global variable</td>
</tr>
</tbody>
</table>

**Return value**
None.

**See also**
- GetGlobalVarID
- GetGlobalVarType
- Get<type>GlobalVar
- IsGlobalVarArray
- IsGlobalVarNull
- IsGlobalVarObject
- Set<type>GlobalVar
**SetMarshaler**

**Description**
Sets a marshaler that will be used to invoke remote methods and convert PowerBuilder data formats to the user’s communication protocol.

**Syntax**
```
SetMarshaler(pbproxyObject obj, IPBX_Marshaler* marshaler)
```

**Argument**
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>An object of type pbproxyObject to be used as a proxy for a remote object that was created using NewProxyObject</td>
</tr>
<tr>
<td>marshaler</td>
<td>A class inherited from IPBX_Marshaler</td>
</tr>
</tbody>
</table>

**Return value**
None.

**Examples**
This example creates a JavaMarshaler class and associates it with a proxy object:

```cpp
// Create JavaMarshaler
JavaMarshaler* marshaler = new JavaMarshaler(env, proxy, jobj);

// Associate the JavaMarshaler with the PB proxy
session->SetMarshaler(proxy, marshaler);

ci->pArgs->GetAt(0)->SetObject(proxy);

ci->returnValue->SetLong(kSuccessful);
return PBX_OK;
```

**Usage**
The `SetMarshaler` function associates an object of type IPBX_Marshaler with a PBProxy object. It is possible to associate multiple marshaler objects with a single proxy object. It is also possible to associate one marshaler object with multiple proxy objects. Neither of these is good coding practice and should be avoided.

Before calling `SetMarshaler`, you can call the IPB_Session GetMarshaler function to obtain an existing marshaler object associated with a given proxy object, and then destroy the existing marshaler object before associating a new marshaler with the proxy.

When a proxy object is destroyed, it calls the associated marshaler object’s Destroy method. If multiple proxy objects are associated with a single marshaler object, you need to implement some form of reference counting. Otherwise, the marshaler object is destroyed when the first associated proxy object is destroyed, and subsequent calls to the marshaler object’s Destroy method, when other associated proxy objects are destroyed, will throw exceptions.
To avoid these issues, there should be a one-to-one relationship between marshaler and proxy objects.

See also
- GetMarshaler
- GetMethodID

SetProp

Description
Adds a new variable to the list of properties of the current session or changes the value of an existing variable.

Syntax
SetProp(LPCTSTR name, void* data)

Return value
None.

Examples
In this example, the native class has two functions. This is their description passed in the PBX_GetDescription function:

"subroutine f_setprop(int a)\n" function int f_getprop()\n"

The functions are associated with these enumerated values:

enum MethodIDs
{
   mid_SetProp = 0,
   mid_GetProp = 1
};

When the f_setprop function is called from PowerBuilder, the following code sets the value of the pointer SetVal to the integer value passed in by f_setprop, then registers that value in the session with the property name prop_name:

int* SetVal = new int;

if (mid == mid_SetProp)
{
   *SetValue = ci -> pArgs -> GetAt(0) -> GetInt();
   session -> SetProp(prop_name, SetVal);
}

Argument | Description
---|---
name | The name of the property to be set
data | A pointer to the data buffer where the variable’s value resides
When the `f_getprop` function is called, the following code uses `GetProp` to set the `GetValue` pointer to point to the value associated with `prop_name`, and then sets the return value to `*GetValue`:

```c
if (mid == mid_GetProp)
{
    int* GetVal;
    GetValue = (int *)session -> GetProp(prop_name);
    ci -> returnValue -> SetInt(*GetVal);
}
```

Usage

SetProp enables you to use a variable value throughout an IPB session without using a global variable, which is susceptible to namespace conflicts with other sessions. SetProp is one of a set of three functions:

- Use SetProp to register a new variable with the session or to change the value of an existing variable.
- Use GetProp to access the variable.
- Use RemoveProp to remove the variable from the list of variables associated with the session when it is no longer needed.

This set of functions is particularly useful for working with multiple threads of execution in EAServer.

Suppose you want to throw an exception from within a PBNI extension and the exception itself is also defined by the PBNI extension. You call the IPB_Session `NewObject` function to create an instance of the exception, causing the PBX_CreateNonVisualObject function to be called.

One way to set the value of the fields of the exception before the function returns in a thread-safe manner is to create a new object or structure to hold the exception information before calling `NewObject`. You can call SetProp to store the structure or the object in the current IPB_Session. When PBX_CreateNonVisualObject is called, you can call GetProp to get the structure or object to obtain the exception information, then call RemoveProp to remove the data you stored in the current session.

See also

GetProp
RemoveProp
SetSharedVarToNull

Description
Sets the value of the specified shared variable to null.

Syntax
SetSharedVarToNull(pbgroup group, pbfield fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group to which the shared variable belongs</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the shared variable</td>
</tr>
</tbody>
</table>

Return value
None.

Examples
This example tests the IsSharedVarNull and SetSharedVarToNull functions:

```powershell
curGroup = session -> GetCurrGroup();
cls = session -> GetClass(myobj);

fid = session -> GetSharedVarID(curGroup, "i_svar");
if (session -> IsSharedVarNull(curGroup, fid))
    session -> SetIntSharedVar(curGroup, fid, 1);
else
    session -> SetSharedVarToNull(curGroup, fid);
```

See also
Get<type>SharedVar
GetSharedVarID
GetSharedVarType
IsSharedVarArray
IsSharedVarNull
IsSharedVarObject
Set<type>SharedVar

SetString

Description
Frees an existing string and assigns a new string value to it by performing a deep copy.

Syntax
SetString (pbstring string, LPCTSTR src)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A valid pbstring variable whose value is to be replaced</td>
</tr>
<tr>
<td>src</td>
<td>The string to be assigned to string</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. Returns PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new string value is invalid; otherwise, returns PBX_E_OUTOF_MEMORY.
Examples

This example uses the IPB_Session SetString method to set the ret_val string to the return value in the PBCallInfo structure. It also uses the IPB_Value SetPBString method to set values in PBCallInfo:

```c
pbclass cls;
pbmethodID mid;
PBCallInfo* ci = new PBCallInfo;
pbstring ret_val;
LPCTSTR pStr;

cls = Session -> GetClass(myobj);
if (isAny)
    mid = Session -> GetMethodID(cls, "uf_any_byvalue",
            PBRT_FUNCTION, "AAAAA");
else
    mid = Session -> GetMethodID(cls, "uf_string_byvalue",
            PBRT_FUNCTION, "SSSSS");
Session -> InitCallInfo(cls, mid, ci);

    ci-> pArgs -> GetAt(0) -> SetPBString(s_low);
    ci-> pArgs -> GetAt(1) -> SetPBString(s_mid);
    ci-> pArgs -> GetAt(2) -> SetPBString(s_high);
    pStr = Session -> GetString(s_null);
    if (pStr != 0)
    {>
        if (strcmp(pStr, "null") == 0 )
            ci-> pArgs -> GetAt(3) -> SetToNull();
        else
            ci-> pArgs -> GetAt(3) -> SetPBString(s_null);
    }
    Session -> InvokeObjectFunction(myobj, mid, ci);
    ret_val = Session -> NewString(");
    Session -> SetPBString(ret_val, Session->GetString
            (ci->returnValue->GetString()));
    Session -> FreeCallInfo(ci);
delete ci;
return ret_val;
```

Usage

A deep copy is performed. The existing value is destroyed first, and then the contents of the src argument are copied into a new value.

See also

NewString
**SetTime**

Description  
Resets the value of the specified pbtime object.

Syntax  
`SetTime (pbtime time, pbint hour, pbint minute, pbdouble second)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>time</code></td>
<td>The pbtime object to be reset</td>
</tr>
<tr>
<td><code>hour</code></td>
<td>An hour in the range 0 to 23</td>
</tr>
<tr>
<td><code>minute</code></td>
<td>A minute in the range 0 to 59</td>
</tr>
<tr>
<td><code>second</code></td>
<td>A second in the range 0 to 59.999999</td>
</tr>
</tbody>
</table>

Return value  
PBX_RESULT. PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new time is invalid.

Examples  
This code puts a new time with the value 01:01:01 into the `time_val` property of the `pArguments` array if the value in the PBCallInfo structure is null. Otherwise it sets `time_val` to the time in the PBCallInfo structure:

```c
if (ci->pArgs->GetAt(i)->IsNull())
{
    pArguments[i].time_val = Session-> NewTime();
    Session->SetTime(pArguments[i].time_val, 1, 1, 1);
    // Time: 01:01:01
}
else
{
    pArguments[i].time_val =
        ci->pArgs->GetAt(i)->GetTime();
}
```

Usage  
If the parameters are invalid, the time is reset to 0:0:0.0.

See also  
NewTime  
SplitTime
CHAPTER 7  PBNI Interfaces, Structures, and Methods

SetValue
Description  Sets the value of one IPB_Value object to the value of another IPB_Value object.
Syntax  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>The value to be replaced</td>
</tr>
<tr>
<td>src</td>
<td>The value to which dest is to be changed</td>
</tr>
</tbody>
</table>

Return value  None.
Examples  These statements set the return value in the PBCallInfo structure ci to the value IPBValue_ret, then release the IBPValue_ret structure:

```
Session -> SetValue(ci -> returnValue, IPBValue_ret);
Session -> ReleaseValue(IPBValue_ret);
```
Usage  Unlike the IPB_Value Set<type> methods, the SetValue method does not verify that the datatype of the replacement value matches the datatype of the original value. The original value is freed and a new value is cloned from the src value. Use this method if you want to swap two different IPB_Value objects that have different types.
See also  AcquireValue, ReleaseValue

SplitDate
Description  Splits the specified pbdate object into a year, month, and day.
Syntax  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>The pbdate object to be split</td>
</tr>
<tr>
<td>year</td>
<td>A year in the range 1000 to 3000</td>
</tr>
<tr>
<td>month</td>
<td>A month in the range 1 to 12</td>
</tr>
<tr>
<td>day</td>
<td>A day in the range 1 to 31</td>
</tr>
</tbody>
</table>

Return value  PBX_RESULT. PBX_OK for success.
Examples  This statement splits the date in the first value in the PBCallInfo structure:

```
Session -> SplitDate(ci-> pArgs -> GetAt(0) ->
    GetDate(), &yy, &mm, &dd);
```
See also  NewDate, SetDate, SplitDateTime
**SplitDateTime**

**Description**
Splits the specified pbdatetime object into a year, month, day, hour, minute, and second.

**Syntax**
`SplitDateTime(pbdatetime dt, pbint *year, pbint *month, pbint *day, pbint *hour, pbint *minute, pbdouble *second)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt</td>
<td>The pbdatetime object to be split</td>
</tr>
<tr>
<td>year</td>
<td>A year in the range 1000 to 3000</td>
</tr>
<tr>
<td>month</td>
<td>A month in the range 1 to 12</td>
</tr>
<tr>
<td>day</td>
<td>A day in the range 1 to 31</td>
</tr>
<tr>
<td>hour</td>
<td>An hour in the range 0 to 23</td>
</tr>
<tr>
<td>minute</td>
<td>A minute in the range 0 to 59</td>
</tr>
<tr>
<td>second</td>
<td>A second in the range 0 to 59.999999</td>
</tr>
</tbody>
</table>

**Return value**
PBX_RESULT. PBX_OK for success.

**See also**
NewDateTime, SetDateTime, SplitDate, SplitTime

**SplitTime**

**Description**
Splits the specified time object into an hour, minute, and second.

**Syntax**
`SplitTime(pbtimate time, pbint *hour, pbint *minute, pbdouble *second)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>The pbtimate object to be split</td>
</tr>
<tr>
<td>hour</td>
<td>An hour in the range 0 to 23</td>
</tr>
<tr>
<td>minute</td>
<td>A minute in the range 0 to 59</td>
</tr>
<tr>
<td>second</td>
<td>A second in the range 0 to 59.999999</td>
</tr>
</tbody>
</table>

**Return value**
PBX_RESULT. PBX_OK for success.

**Examples**
These statements split a time into hours, minutes, and seconds, and then use the resulting values to set the value of a new time object:

```pascal
Session->SetTime(ret_val, hh, mm, ss);
```

**See also**
NewTime, SetTime
CHAPTER 7  PBNI Interfaces, Structures, and Methods

**ThrowException**

**Description**
Throws a PowerBuilder exception or inherited exception, and replaces the existing exception if there is one.

**Syntax**
ThrowException (pbobject ex)

**Return value**
None.

**Examples**
This code creates a new exception object in the class `user_exception_pspp`, invokes its `setMessage` function, and throws the exception:

```c
pbclass cls;
pbmethodID mid;
pbobject ex;
pbgroup group;
PBCallInfo* ci = new PBCallInfo;

// Throw exception
group = session->FindGroup("user_exception_pspp", pbgroup_userobject);
if (group==NULL)
    return;
cls = session->FindClass(group, "user_exception_pspp");
if (group==NULL)
    return;
ex = session -> NewObject(cls);
mid = session-> GetMethodID(cls, "setmessage", PBRT_FUNCTION, "QS");
session-> InitCallInfo(cls,mid,ci);

ex-> SetPBString(session, "Test exception");
mid = session-> GetMethodID(cls, "setmessage", PBRT_FUNCTION, "QS");
session-> InitCallInfo(cls,mid,ci);

session-> InvokeObjectFunction(ex,mid,ci);
if (!ThrowToPB)
    session -> ClearException();
session -> FreeCallInfo(ci);
deleci;

See also
ClearException
GetException
HasExceptionThrown
```
**TriggerEvent**

**Description**
Triggers a PowerBuilder event.

**Syntax**
```
TriggerEvent(pbobject obj, pbmethodID mid, PBCallInfo * ci)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj</code></td>
<td>The pbobject containing the method</td>
</tr>
<tr>
<td><code>mid</code></td>
<td>The pbMethodID returned by GetMethodID</td>
</tr>
<tr>
<td><code>ci</code></td>
<td>A pointer to a preallocated PBCallInfo structure</td>
</tr>
</tbody>
</table>

**Return value**
PBXRESULT. Returns PBX_OK for success, or one of the following for failure:
- PBX_E_INVALID_ARGUMENT
- PBX_E_INVOKE_METHOD_INACCESSIBLE
- PBX_E_INVOKE_WRONG_NUM_ARGS
- PBX_E_INVOKE_REFARG_ERROR
- PBX_E_INVOKE_METHOD_AMBIGUOUS
- PBX_E_INVOKE_FAILURE

**Examples**
This code triggers the clicked event on a DataWindow object:
```
cls = session->GetClass(dwobj);
mid = session->GetMethodID
   (cls, "clicked", PBRT_EVENT, "LIILCdwobject.");
session->InitCallInfo(cls, mid, ci);
session->TriggerEvent(dwobj, mid, ci);
...```

**See also**
GetClass
GetMethodID
**UpdateField**

**Description**
Refreshes a visual property of a PowerBuilder object.

**Syntax**
```
UpdateField(pbobject obj, pbfieldID fid)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>obj</em></td>
<td>The <em>pbobject</em> whose user interface property needs to be changed</td>
</tr>
<tr>
<td><em>fid</em></td>
<td>The field ID of the object</td>
</tr>
</tbody>
</table>

**Return value**
PBXRESULT. Returns success or failure.

**Examples**
This function changes the title of a DataWindow control:

```c
void CallBack::f_newtitle(IPB_Session* session, pbstring str_val, pbobject dwobj)
{
    pbclass cls;
    pbfieldID fid;
    cls=session->GetClass(dwobj);
    fid=session->GetFieldID(cls, "title");
    if (fid==kUndefinedFieldID)
        return;
    session -> SetPBStringField(dwobj,fid,str_val);
    session -> UpdateField(dwobj,fid);
    return ;
}
```

**Usage**
When you change any visual property of a PowerBuilder object by calling `Set<type>field` functions, the property is changed but the property is not refreshed in the graphical user interface. `UpdateField` refreshes the visual properties of PowerBuilder objects. You must call this function explicitly when changing any visual property with the `Set<type>field` functions.

**See also**
`Set<type>Field`
**IPB_Value interface**

**Description**

The IPB_Arguments and IPB_Value interfaces pass values between the PowerBuilder VM and PowerBuilder extension modules. Through the IPB_Value interface, you can access information about each variable, including its type, null flag, access privileges, array or simple type, and reference type.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get&lt;type&gt;</td>
<td>Set of datatype-specific methods that return a pointer to the data in IPB_Value</td>
</tr>
<tr>
<td>GetClass</td>
<td>Returns the class handle of a PowerBuilder object</td>
</tr>
<tr>
<td>GetType</td>
<td>Returns the datatype of a single data item or array</td>
</tr>
<tr>
<td>IsArray</td>
<td>Returns true if the IPB_Value instance contains an array, otherwise false</td>
</tr>
<tr>
<td>IsByRef</td>
<td>Returns true if the IPB_Value instance is passed by reference</td>
</tr>
<tr>
<td>IsEnum</td>
<td>Returns true if the IPB_Value instance contains a null value, otherwise false</td>
</tr>
<tr>
<td>IsObject</td>
<td>Returns true if the IPB_Value instance contains an object or object array, otherwise false</td>
</tr>
<tr>
<td>SetToNull</td>
<td>Used to set the data contained in the IPB_Value instance to null so that data can be reset</td>
</tr>
<tr>
<td>Set&lt;type&gt;</td>
<td>Set of datatype-specific methods that set the value of the IPB_Value instance</td>
</tr>
</tbody>
</table>
Get<type>

Description
A set of datatype-specific methods that return a pointer to the data in IPB_Value.

Syntax
GetArray ( )
GetBlob ( )
GetBool ( )
GetByte ( )
GetChar ( )
GetDate ( )
GetDateTime ( )
GetDecimal ( )
GetDouble ( )
GetInt ( )
GetLong ( )
GetLongLong ( )
GetObject ( )
GetReal ( )
GetString ( )
GetTime ( )
GetUlong ( )

Return value
A predefined PBNI datatype that corresponds to the PowerBuilder datatype in the method name.

Examples
This statement gets the date in the first value in the PBCallInfo structure and splits it into year, month, and day:

```
Session -> SplitDate(ci-> pArgs -> GetAt(0) ->
    GetDate(),&yy,&mm,&dd);
```

Usage
If IPB_Value contains a null value, or if you are trying to get a specific datatype from an IPB_Value instance of another datatype, the data retrieved is undetermined. If the datatype is string, blob, decimal, time, date, datetime, array, or object, the return value points to the same address pointed to by IPB_Value. As a result, changing either the variable that holds the return value or the value of the IPB_Value instance affects the other.

See also
Set<type>
**GetClass**

Description: Returns the class handle of a PowerBuilder object.

Syntax: GetClass()

Return value: pbclass or null on error.

Examples:
```
    pbclass clz = ci->pArgs->GetAt(i)->GetClass();
```

See also: Get<type>, GetType, Set<type>

---

**GetType**

Description: Returns the datatype of a single data item or array.

Syntax: GetType()

Return value: pbuint

Examples:
```
    ArgsType = ci->pArgs->GetAt(i)->GetType();
    switch (ArgsType)
    {
        case pbvalue_int:
            if (ci->pArgs->GetAt(i)->IsNull())
                pArguments[i].int_val=1;
            else
                pArguments[i].int_val =
                    ci->pArgs->GetAt(i)->GetInt();
            break;
        ...
```

Usage: If the IPB_Value instance contains an object or structure, GetType returns the class ID of the data. Otherwise, it returns a simple datatype defined in the list of pbvalue_type enumerated types.

See also: Get<type>, GetClass, Set<type>
**IsArray**

**Description**
Returns true if the IPB_Value instance contains an array; otherwise, returns false.

**Syntax**
IsArray()

**Return value**
pbboolean

**Examples**
This example tests whether an IPB_Value instance is an array before obtaining the array:

```c
if (ci->pArgs->GetAt(i)->IsArray())
{
    pArguments[i].array_val =
    ci->pArgs->GetAt(i)->GetArray();
    continue;
}
```

**See also**
IsByRef
IsEnum
IsObject

**IsByRef**

**Description**
Returns true if the IPB_Value instance contains a by reference argument; otherwise it returns false.

**Syntax**
IsByRef()

**Return value**
pbboolean

**Examples**
This example shows how you would use IsByRef to test whether an argument is obtained by reference:

```c
if (ci->pArgs->GetAt(i)->IsByRef())
    ...
```

**See also**
IsArray
IsEnum
IsObject
IPB_Value interface

IsEnum
Description
Returns true if the IPB_Value instance contains an enumerated value; otherwise it returns false.
Syntax
IsEnum()
Return value
pbboolean
See also
GetEnumItemName
GetEnumItemValue

IsNull
Description
Returns true if the IPB_Value instance contains a null value; otherwise, it returns false.
Syntax
IsNull()
Return value
pbboolean
Examples
This example tests whether an IPB_Value instance contains a null value before attempting to obtain its value:

```c
if (ci->pArgs->GetAt(i)->IsObject())
{
    if (ci->pArgs->GetAt(i)->IsNull())
        pArguments[i].obj_val = 0;
    else
        pArguments[i].obj_val = ci->pArgs->GetAt(i)->GetObject();
    continue;
}
```

See also
IsArray
IsByRef
IsObject
SetToNull
**IsObject**

**Description**
Returns true if the IPB_Value instance contains an object or object array; otherwise it returns false.

**Syntax**
IsObject()

**Return value**
pbboolean

**Examples**
This example tests whether an IPB_Value instance contains an object before attempting to obtain the object:

```c
if( ci->pArgs->GetAt(i)->IsObject() )
{
    if (ci->pArgs->GetAt(i)->IsNull())
        pArguments[i].obj_val = 0;
    else
        pArguments[i].obj_val =
            ci->pArgs->GetAt(i)->GetObject();
    continue;
}
...```

**See also**
IsArray
IsByRef
IsEnum
**IPB_Value interface**

**Set<type>**

Description: Set of datatype-specific methods that set the value of the IPB_Value instance.

Syntax:
- `SetArray ( pbarray array )`
- `SetBlob( pbblob blob )`
- `SetBool ( pbboolean boolean )`
- `SetByte ( pbbyte byte )`
- `SetChar ( pbchar char )`
- `SetDate ( pbdate date )`
- `SetDateTime( pbdatetime datetime )`
- `SetDecimal ( pbdecimal dec)`
- `SetDouble ( pbdouble double)`
- `SetInt ( pbint int )`
- `SetLong( pblong long )`
- `SetLongLong( pblonglong longlong )`
- `SetObject ( pbobject object )`
- `SetPBString ( pbstring string)`
- `SetReal( pbreal real )`
- `SetString ( LPCTSTR string)`
- `SetTime( pbtime time )`
- `SetUint( pbuint uint )`
- `SetUlong ( pbulong ulong )`

Return value: PBXRESULT.
Examples

This example uses the IPB_Value SetPBString method to set values in PBCallInfo. It also uses the IPB_Session SetString method to set the ret_val string to the return value in the PBCallInfo structure:

```c
pbclass cls;
pbmethodID mid;
PBCallInfo* ci = new PBCallInfo;
pbstring ret_val;
LPCTSTR pStr;

ci = Session -> GetClass(myobj);
if (isAny)
    mid=Session-> GetMethodID(cls, "uf_any_byvalue",
        PBRT_FUNCTION, "AAAAA");
else
    mid=Session-> GetMethodID(cls, "uf_string_byvalue",
        PBRT_FUNCTION, "SSSSS");
Session-> InitCallInfo(cls, mid, ci);

// Call IPB_Value SetPBString method
if (pStr != 0)
{
    if (strcmp(pStr, "null") == 0 )
        ci-> pArgs -> GetAt(3) -> SetToNull();
    else
        ci-> pArgs -> GetAt(3) -> SetPBString(s_null);
}

Session -> InvokeObjectFunction(myobj, mid, ci);
ret_val = Session -> NewString("");

// Call IPB_Session SetString method
Session -> SetString(ret_val, Session->GetString
    (ci->returnValue->GetString()));
Session -> FreeCallInfo(ci);
delete ci;
return ret_val;
```
**IPB_Value interface**

**Usage**
These methods automatically set the value of IPB_Value to not null and return an error if the datatype to be set does not match the existing datatype. The error code is PBX_E_MISMATCHED_DATA_TYPE. If the value is a read-only argument, it returns the error PBX_E_READONLY_ARGS. If the datatype is string or blob, a deep copy is performed. The existing value is destroyed first, and then the contents of the argument are copied into a new value.

**See also**
Get<type>

**SetToNull**

**Description**
Sets the data contained in the IPB_Value instance to null so the data can be reset.

**Syntax**
SetToNull()

**Return value**
PBXRESULT. If the value is a read-only argument, the error PBX_E_READONLY_ARGS is returned.

**Examples**
This example shows the use of SetToNull when a null blob value is returned:

```c
    case pbvalue_blob:
      pStr=(LPCSTR)Session->GetBlob(retVal.blob_val);
      if (strncmp(pStr, "null", 4)==0 )
        ci->returnValue->SetToNull();
      else
        ci->returnValue->SetBlob(retVal.blob_val);

      break;
...```

**See also**
IsEnum
IPB_VM interface

Description
The IPB_VM interface loads PowerBuilder applications in third-party applications and interoperates with the PowerBuilder virtual machine (PBVM).

Methods
IPB_VM has two methods:
- CreateSession
- RunApplication

CreateSession

Description
Creates an IPB_Session object that can be used to call PowerBuilder functions.

Syntax
CreateSession(LPCTSTR applicationName, LPCTSTR* libraryList, pbuint numLibs, IPB_Session** session)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationName</td>
<td>The name of the current application object in lowercase</td>
</tr>
<tr>
<td>libraryList</td>
<td>The library list of the PowerBuilder application that contains the objects and functions to be called</td>
</tr>
<tr>
<td>numLibs</td>
<td>The number of libraries in the library list</td>
</tr>
<tr>
<td>session</td>
<td>A pointer to IPB_Session*, which will return the current IPB_Session pointer after the call</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. PBX_OK for success.
Examples

This example creates an IPB_Session with the simple library list `mydemo.pbl`:

```c
IPB_Session* session;
IPB_VM* vm = NULL;
fstream out;
ifstream in;
PBXRESULT ret;

HINSTANCE hinst=LoadLibrary("pbvm125.dll");
if ( hinst== NULL) return 0;

out<< "Loaded PowerBuilder VM successfully!"<<endl;

P_PB_GetVM getvm = (P_PB_GetVM)GetProcAddress(hinst, "PB_GetVM");
if (getvm == NULL) return 0;

getvm(&vm);
if (vm == NULL) return 0;

static const char *liblist[] =
{
  "mydemo.pbl"
};

ret= vm->CreateSession("mydemo", liblist, 1, &session);
if (ret != PBX_OK)
{
  out << "Create session failed." << endl;
  return 0;
}
out << "Create session succeeded!" <<endl;
```

See also

RunApplication
RunApplication

Description
Runs the specified application.

Syntax
RunApplication(LPCTSTR applicationName, LPCTSTR* libraryList, pbuint numLibs, LPCSTR commandLine, IPB_Session** session)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationName</td>
<td>The name of the application object to be run, in lowercase</td>
</tr>
<tr>
<td>libraryList</td>
<td>The library list of the application</td>
</tr>
<tr>
<td>numLibs</td>
<td>The number of libraries in the library list</td>
</tr>
<tr>
<td>commandLine</td>
<td>Parameters to be passed to the application object</td>
</tr>
<tr>
<td>session</td>
<td>A pointer to IPB_Session*, which will return the current IPB_Session pointer after the call</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. PBX_OK for success.

Examples
This code fragment loads the PowerBuilder VM and runs an application called runapp that uses one library, runapp.pbd. It passes in a command line with two arguments:

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
  LPCTSTR szHello = "Hello world"

  // Provide command line parameters (employee ids)
  // to be passed to the PowerBuilder application
  LPCTSTR szcommandline = "102 110"

  int wmId, wmEvent, ret;
  PAINTSTRUCT ps;
  HDC hdc;

  switch (message)
  {
    case WM_CREATE:
    {
      hPBVMInst = ::LoadLibrary("pbvm125.dll");

      P_PB_GetVM getvm = (P_PB_GetVM)
        GetProcAddress(hPBVMInst,"PB_GetVM");

      IPB_VM* vm = NULL;

      P_BGetVM runapp = getvm("runapp.pbd");
      runapp->Run(ApplicationName, szcommandline, 1, ret);
    }
  }
}
getvm(&vm);

static const char *liblist [] =
    {"runapp.pbd");

vm->RunApplication("runapp", liblist, 1,
    szcommandline, &session);

break;
}

See also CreateSession
IPBX_Marshaler interface

Description
The IPBX_Marshaler interface is used to invoke remote methods and convert PowerBuilder data formats to the user’s communication protocol. A marshaler extension is a PowerBuilder extension that acts as the bridge between PowerBuilder and other components, such as EJBs, Java classes, CORBA objects, Web services, and so on.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destroy</td>
<td>Destroys an instance of an object inherited from the IPBX_Marshaler structure</td>
</tr>
<tr>
<td>GetModuleHandle</td>
<td>Returns the handle of the PBX that contains the native class</td>
</tr>
<tr>
<td>InvokeRemoteMethod</td>
<td>Used in PowerBuilder marshaler native classes to call remote methods</td>
</tr>
</tbody>
</table>

Destroy

Description
Use the Destroy method to destroy instances of objects inherited from the IPBX_Marshaler structure.

Syntax
Destroy()

Return value
None.

Examples
This code destroys the current instance of the SampleMarshaler structure:

```cpp
void SampleMarshaler::Destroy()
{
    delete this;
}
```

Usage
You must implement this method in the marshaler native class after creating an instance of a marshaler structure and invoking remote methods.

See also
GetModuleHandle
InvokeRemoteMethod
GetModuleHandle

Description
Returns the handle of the PBX that contains the native class. This method is required to allow the PowerBuilder VM to determine which PBXs can be unloaded.

Syntax
GetModuleHandle()

Return value
pbulong

Examples
This code in the implementation of a marshaler class returns the handle of the PBX:

extern pbulong thisModuleHandle;
ppbulong SampleMarshaler::GetModuleHandle()
{
    return thisModuleHandle;
}

The handle is set in the main module:

pbulong thisModuleHandle = 0;

BOOL APIENTRY DllMain(HANDLE hModule,
    DWORD ul_reason_for_call,
    LPVOID lpReserved
)
{
    thisModuleHandle = (pbulong)hModule;

    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}

Usage
You must implement this method in the marshaler native class.

See also
Destroy
InvokeRemoteMethod
InvokeRemoteMethod

Description
Used in PowerBuilder marshaler native classes to call remote methods.

Syntax
InvokeRemoteMethod(IPB_Session *session, pbproxyobject obj, LPCTSTR methodDesc, PBCallInfo *ci)

Argument | Description
---|---
session | This IPB session
obj | The proxy object for the remote object
methodDesc | An arbitrary string stored as an alias name for the remote method in the proxy, for example:
function int foo(int a) alias "This is a method in remote BizTalk"
ci | The parameters and return value setting for the call

Return value
PBXRESULT.PBX_OK if the call succeeded.

Examples
This example shows a header file for a sample marshaler class:

```c
#include "sampleinclude.h"
#include <pbext.h>

class SampleMarshaler : public IPBX.Marshaler
{
private:
    string   d_mystring;
    long     d_mylong;

private:
    void myMethod(string arg1);

public:
    SampleMarshaler(
        string myString,
        long   mylong
    );
    ~SampleMarshaler();

    virtual PBXRESULT InvokeRemoteMethod
    (            
        IPB_Session*   session,
        pbproxyObject  obj,
        LPCTSTR        methodDesc,
        PBCallInfo*    ci
    );
```
IPBX_NonVisualObject interface

```cpp
    virtual pbulong GetModuleHandle();
    virtual void Destroy();
};
```

The associated C++ implementation file contains code like this:

```cpp
PBXRESULT SampleMarshaler::InvokeRemoteMethod
{
    IPB_Session* session,
    pbproxyObject obj,
    LPCTSTR methodDesc,
    PBCallInfo* ci
}
{
    // method invocation
}
```

Usage

You must implement this method in the marshaler native class.

See also

Destroy
GetModuleHandle

IPBX_NonVisualObject interface

Description

The IPBX_NonVisualObject interface inherits from IPBX_UserObject and is the direct ancestor class of nonvisual PowerBuilder native classes.

Methods

IPBX_NonVisualObject inherits two methods from the IPBX_UserObject interface: Destroy and Invoke.
**IPBX_UserObject interface**

**Description**

The IPBX_UserObject interface is the ancestor class of the PowerBuilder native classes.

**Methods**

IPBX_UserObject has two methods: Destroy and Invoke.

---

**Destroy**

**Description**

Destroys the current instance of a PowerBuilder native class that inherits from IPBX_UserObject.

**Syntax**

Destroy()

**Return value**

None.

**Examples**

This example shows how you would call Destroy for the class MyPBNIClass:

```cpp
void MyPBNIClass::Destroy()
{
    delete this;
}
```

**Usage**

You must implement this method in the native class after creating an instance of the class and invoking remote methods.

**See also**

Invoke
Invoke

Calls methods in PowerBuilder native classes.

Syntax

Invoke(IPB_Session * session, pboject obj, pbmethodID mid, PBCallInfo * ci)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>session</td>
<td>This IPB session</td>
</tr>
<tr>
<td>obj</td>
<td>The PowerBuilder extension object to be invoked</td>
</tr>
<tr>
<td>mid</td>
<td>The pbMethodID returned by GetMethodID</td>
</tr>
<tr>
<td>ci</td>
<td>The parameters and return value setting for the call</td>
</tr>
</tbody>
</table>

Return value

PBXRESULT.PBX_OK for success.

Examples

In this example, the method invoked depends on the value (0, 1, or 2) of the method ID returned from the GetMethodID method:

```c
PBXRESULT PBNIExt::Invoke
{
    IPB_Session  *session,
    pboject     obj,
    pbmethodID  mid,
    PBCallInfo  * ci
}
{
    PBXRESULT result = PBX_OK;

    switch (mid)
    {
        case mFuncA:
            result = FuncA(session, obj, ci);
            break;
        case mFuncB:
            result = FuncB(session, obj, ci);
            break;
        case mFuncC:
            result = FuncC(session, obj, ci);
            break;
        default:
            result = PBX_E_INVOKE_FAILURE;
            break;
    }

    return PBX_OK;
}
```

See also

GetMethodID
**IPBX_VisualObject interface**

**Description**

The IPBX_VisualObject interface inherits from IPBX_UserObject and is the direct ancestor class of visual PowerBuilder native classes.

**Methods**

IPBX_VisualObject has three direct methods:

- CreateControl
- GetEventID
- GetWindowClassName.

IPBX_NonVisualObject inherits two methods from the IPBX_UserObject interface:

- Destroy
- Invoke

**CreateControl**

**Description**

Creates a window control and returns its handle to the PowerBuilder VM.

**Syntax**

CreateControl(DWORD dwExStyle, LPCTSTR lpWindowName, DWORD dwStyle, int x, int y, int nWidth, int nHeight, HWND hWndParent, HINSTANCE hInstance)

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwExStyle</td>
<td>The extended window style</td>
</tr>
<tr>
<td>lpWindowName</td>
<td>The window name</td>
</tr>
<tr>
<td>dwStyle</td>
<td>The window style</td>
</tr>
<tr>
<td>x</td>
<td>The horizontal position of the window</td>
</tr>
<tr>
<td>y</td>
<td>The vertical position of the window</td>
</tr>
<tr>
<td>nWidth</td>
<td>The window's width</td>
</tr>
<tr>
<td>nHeight</td>
<td>The window's height</td>
</tr>
<tr>
<td>hWndParent</td>
<td>The handle of the parent or owner window</td>
</tr>
<tr>
<td>hInstance</td>
<td>The handle of the application instance</td>
</tr>
</tbody>
</table>

**Return value**

HWND.

**Examples**

This is part of a visual extension example available on the Sybase Web site:

```cpp
LPCTSTR CVisualExt::GetWindowClassName()
{
    return s_className;
}
```

```cpp
HWND CVisualExt::CreateControl
```
IPBX_VisualObject interface

{
    DWORD dwExStyle,      // extended window style
    LPCTSTR lpWindowName, // window name
    DWORD dwStyle,   // window style
    int x,           // horizontal position of window
    int y,           // vertical position of window
    int nWidth,      // window width
    int nHeight,     // window height
    HWND hWndParent, // handle to parent or owner window
    HINSTANCE hInstance   // handle to application
    instance
}
{
    d_hwnd = CreateWindowEx(dwExStyle, s_className,
                            lpWindowName, dwStyle, x, y, nWidth, nHeight,
                            hWndParent, NULL, hInstance, NULL);

    ::SetWindowLong(d_hwnd, GWL_USERDATA, (LONG)this);

    return d_hwnd;
}

Usage
The window must be registered before you call CreateControl.

See also
GetEventID
GetWindowClassName
PBNI Programmers Guide and Reference

CHAPTER 7  PNI Interfaces, Structures, and Methods

**GetEventID**

Returns the identifier of an event when the window’s parent is notified that the event occurred.

**Syntax**

GetEventID(HWND hWnd, uint iMsg, WPARAM wParam, LPARAM lParam)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hWnd</td>
<td>The handle of the parent window.</td>
</tr>
<tr>
<td>iMsg</td>
<td>The message sent to the parent.</td>
</tr>
</tbody>
</table>
| wParam   | The word parameter of the message. For WM_COMMAND, the high-order word specifies:  
  - The notification code if the message is from a control  
  - 1 if the message is from an accelerator  
  - 0 if the message is from a menu.  
  The low-order word specifies the identifier of the control, accelerator, or menu.  
For WM_NOTIFY, this parameter contains the identifier of the control sending the message. |
| lParam   | The long parameter of the message. For WM_COMMAND, this parameter contains the handle of the control sending the message if the message is from a control. Otherwise, this parameter is null.  
For WM_NOTIFY, this parameter contains a pointer to a structure. |

**Return value**  
Integer.

**Examples**

In this example, the GetEventID function returns the identifier PB_BNCLICKED if a WM_COMMAND message with the notification code BN_CLICKED was sent. It returns the identifier PB_ENCHANGE if a WM_NOTIFY message was sent; otherwise it returns PB_NULL.

```cpp
TCHAR CVisualExt::s_className[] = "PBVisualExt";

LPCTSTR CVisualExt::GetWindowClassName()
{
    return s_className;
}

HWND CVisualExt::CreateControl
{ (PBN_PROGRAMMERS_GUIDE, REFERENCE)
IPBX_VisualObject interface

DWORD dwExStyle, // extended window style
LPCTSTR lpWindowName, // window name
DWORD dwStyle, // window style
int x, // horizontal position of window
int y, // vertical position of window
int nWidth, // window width
int nHeight, // window height
HWND hWndParent, // handle of parent or owner window
HINSTANCE hInstance // handle of application instance

{ 
d_hwnd = CreateWindowEx(dwExStyle, s_className,
lpWindowName, dwStyle, x, y, nWidth, nHeight,
hWndParent, NULL, hInstance, NULL);

::SetWindowLong(d_hwnd, GWL_USERDATA, (LONG)this);

return d_hwnd;
}

int CVisibleExt::GetEventID(
HWND hWnd, /* Handle of parent window */
UINT iMsg, /* Message sent to parent window*/
WPARAM wParam, /* Word parameter of message*/
LPARAM lParam /* Long parameter of message*/
)

{ 
if (iMsg == WM_COMMAND)
{
if ((HWND)lParam == d_hwnd)
{
    switch(HIWORD(wParam))
    {
    case BN_CLICKED:
        return PB_BNCLICKED;
        break;
    }
}
}

if (iMsg == WM_NOTIFY)
{
    return PB_ENCHANGE;
}

return PB_NULL;
}
Usage
This function is used to process Windows messages, such as WM_COMMAND and WM_NOTIFY, that are sent to the parent of an object and not to the object itself. Such messages cannot be caught in the visual extension’s window procedure. The PBVM calls GetEventID to process these messages.

If the message is mapped to a PowerBuilder event, GetEventID returns the event’s identifier, for example PB_BNCLICKED, and the event is fired automatically. PowerBuilder event token identifiers are mapped to unsigned integer values in the pbevtid.h header file. The identifiers in pbevtid.h are associated with PowerBuilder event token names. For example, the identifier PB_BNCLICKED is associated with the token name pbm_bnclicked.

If the message is not mapped to an event, GetEventID returns the value PB_NULL and the message is discarded.

See also
CreateControl
GetWindowClassName

GetWindowClassName
Description
Returns the name of the window.
Syntax
GetWindowClassName()
Return value
LPCTSTR.
Examples
The string returned by GetWindowClassName is passed as an argument to the CreateControl method:

```
LPCTSTR CVirtualExt::GetWindowClassName()
{
   return s_className;
}
```

Usage
The window must be registered before you call GetWindowClassName.

See also
CreateControl
GetEventID
PBArrayInfo structure

Description
PBArrayInfo is a C++ structure used to hold information about arrays.

Properties

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayBound</td>
<td>Local struct declaration</td>
<td>Structure of type pblong containing the boundaries (upperBound, lowerBound) of a dimension.</td>
</tr>
<tr>
<td>BoundedArray</td>
<td>Enum data</td>
<td>Used in arrayType to identify that the array is a bounded array.</td>
</tr>
<tr>
<td>UnboundedArray</td>
<td>Enum data</td>
<td>Used in arrayType to identify that the array is an unbounded array.</td>
</tr>
<tr>
<td>arrayType</td>
<td>Enum type</td>
<td>Used in IPB_Session::GetArrayInfo to identify the datatype of the array. Do not set this variable manually.</td>
</tr>
<tr>
<td>valueType</td>
<td>pbuint</td>
<td>The datatype of array items. Set it to pbvalue_type if it is a simple type, or pbobject if the item is a class or structure.</td>
</tr>
<tr>
<td>numDimensions</td>
<td>pbuint</td>
<td>Number of dimensions of the array. An unbounded array can have only one dimension. The lower bound is one.</td>
</tr>
<tr>
<td>bounds</td>
<td>ArrayBound[]</td>
<td>Array bounds declaration array, used in a bounded array.</td>
</tr>
</tbody>
</table>

PBCallInfo structure

Description
PBCallInfo is a C++ structure used to hold arguments and return type information in function calls between PBNI and PowerBuilder.

Properties

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pArgs</td>
<td>IPB_Arguments*</td>
<td>Interface used to access arguments</td>
</tr>
<tr>
<td>returnType</td>
<td>IPB_Value</td>
<td>Holds return data after the call</td>
</tr>
<tr>
<td>returnClass</td>
<td>pbclass</td>
<td>Holds return class after the call</td>
</tr>
</tbody>
</table>
PB_DateData structure

Description: The PB_DateData structure is used to pass data of type Date in the SetData function in the IPB_RSItemData interface.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>A short identifying the year</td>
</tr>
<tr>
<td>month</td>
<td>A short identifying the month</td>
</tr>
<tr>
<td>day</td>
<td>A short identifying the day</td>
</tr>
<tr>
<td>filler</td>
<td>A short used for structure alignment only</td>
</tr>
</tbody>
</table>

See also: SetData

PB_DateTimeData structure

Description: The PB_DateTimeData structure is used to pass data of type DateTime in the SetData function in the IPB_RSItemData interface.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>A PB_DateData structure identifying the date</td>
</tr>
<tr>
<td>time</td>
<td>A PB_TimeData structure identifying the time</td>
</tr>
</tbody>
</table>

See also: SetData

PB_TimeData structure

Description: The PB_TimeData structure is used to pass data of type Time in the SetData function in the IPB_RSItemData interface.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hour</td>
<td>A short identifying the hour</td>
</tr>
<tr>
<td>minute</td>
<td>A short identifying the minute</td>
</tr>
<tr>
<td>second</td>
<td>A short identifying the second</td>
</tr>
<tr>
<td>filler</td>
<td>A short used for structure alignment only</td>
</tr>
</tbody>
</table>

See also: SetData
PBX_DrawItemStruct structure

Description
The PBX_DrawItemStruct structure contains the properties of an external visual control that you want to draw using the PBX_DrawVisualObject function.

Table 7-11: PBX_DrawItemStruct members

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>X coordinate of the visual control relative to its parent control (for example, the window that contains it).</td>
</tr>
<tr>
<td>y</td>
<td>Y coordinate of the visual control relative to its parent control.</td>
</tr>
<tr>
<td>width</td>
<td>Width of the visual control.</td>
</tr>
<tr>
<td>height</td>
<td>Height of the visual control.</td>
</tr>
<tr>
<td>objectName</td>
<td>The name of the visual object, for example: uo_1.</td>
</tr>
<tr>
<td>tag</td>
<td>Field to be used to pass any value at the user’s discretion.</td>
</tr>
<tr>
<td>enabled</td>
<td>Whether the visual control is enabled. Possible values are true and false.</td>
</tr>
<tr>
<td>visible</td>
<td>Whether the visual control is visible. Possible values are true and false. In the development environment, PowerBuilder does not call the PBX_DrawVisualObject function if this field is set to false and the Design&gt; Show Invisibles menu item is not selected.</td>
</tr>
<tr>
<td>borderstyle</td>
<td>Border style of the visual control. A value of the pbborder_style enumerated variable. Possible values are:</td>
</tr>
<tr>
<td></td>
<td>• 0 – none</td>
</tr>
<tr>
<td></td>
<td>• 1 – shadowbox</td>
</tr>
<tr>
<td></td>
<td>• 2 – box</td>
</tr>
<tr>
<td></td>
<td>• 5 – lowered</td>
</tr>
<tr>
<td></td>
<td>• 6 – raised</td>
</tr>
<tr>
<td>backgroundColor</td>
<td>Background color of the visual control. You can obtain the RGB value of the background color using the Windows API functions GetRValue, GetGValue, and GetBValue.</td>
</tr>
</tbody>
</table>

See also
PBX_DrawVisualObject
PBArrayAccessor template class

Description

There are two versions of the PBArrayAccessor template class. The first version is used to access the items in an array of a standard type. The second version is used to access items in a string array. The standard types are defined as ValueTypes in pbtraits.h and are pbint, pbuint, pbbyte, pblong, pblonglong, pbulong, pbboolean, pbreal, pbdouble, pbdec, pbdate, pbtime, pbdatetime, pbchar, pbblob, and pbstring.

PBArrayAccessor has four methods:

- GetAt
- IsNull
- SetAt
- SetToNull

GetAt

Description

Obtains the array item at the specified dimension.

Syntax

GetAt(pblong dim[])

Return value

ValueType (defined in pbtraits.h).

Argument Description

dim The dimension of the array item to be obtained

Examples

See SetAt.

See also

SetAt

IsNull

Description

Returns true if the array item contains a null value, otherwise returns false.

Syntax

IsNull(pblong dim[])

Argument Description

dim The dimension of the array item to be tested

Return value

pbboolean.

See also

GetAt, SetAt, SetToNull
**PBArraryAccessor template class**

**SetAt**

**Description**
Sets the array item at the specified dimension.

**Syntax**
For arrays of a specified ValueType:

\[
\text{SetAt}(\text{pblong dim[ ], ValueType \( v \)})
\]

For string arrays:

\[
\begin{align*}
\text{SetAt}(\text{pblong dim[ ], LPCTSTR string}) \\
\text{SetAt}(\text{pblong dim[ ], pbstring string})
\end{align*}
\]

**Argument** | **Description**
--- | ---
\( dim \) | The dimension of the array item to be set
\( v \) | A ValueType defined in `pbtraits.h`
\( string \) | A string of type `pbstring` or `LPCTSTR`

**Return value**
None.

**Examples**
This example shows the use of `GetAt` and `SetAt` in arrays of a type specified by a `Value`:

```cpp
template < typename T, pbvalue_type I>
void ArrayCreator<T, I>::f_unbounded_simple_array(
    IPB_Session* session,
    ifstream in,
    ofstream out,
    LPCSTR data_type)
{
    pbarray out_array;
    int i;
    pblong dim[4], itemcount1, itemcount2;
    T *iarg, oarg;

    in >> itemcount1;
    iarg = new T[itemcount1];
    // Create unbounded integer array
    { 
        PBUnboundedArrayCreator<I> ac(session);
        out_array = ac.GetArray();
        PBArrayAccessor<I> aa(session, out_array);
        for (i=0; i<itemcount1; i++)
            in >> iarg[i];
        for (i=0; i<itemcount1; i++)
            { 
                dim[0]=i+1;
                aa.SetAt(dim, iarg[i]);
```
itemcount2 = session->GetArrayItemCount(out_array);
out << "The array item count is " << itemcount2 << endl;
for (i=0; i<itemcount2; i++)
{
    dim[0]=i+1;
    oarg=aa.GetAt(dim);
    if (oarg != iarg[i])
        out << "*** ERROR" << endl;
    else
        out << oarg << " ";
}
out << endl;
delete []iarg;
out << endl;
return;

See also  GetAt

### SetToNull

**Description**
Sets the value of the specified array item to null.

**Syntax**
SetToNull(pblong *dim[])

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>The dimension of the array item to be set</td>
</tr>
</tbody>
</table>

**Return value**
None.

**See also**
GetAt
IsNull
SetAt
PBBoundedArrayCreator template class

There are two versions of the PBBoundedArrayCreator template class. The first version is used to create a bounded array of a standard type. The standard types are defined as ValueTypes in pbtraits.h and are pbint, pbuint, pbbyte, pblong, pblonglong, pbulong, pbboolean, pbreal, pbdouble, pbdec, pbdate, pbtime, pbdatetime, pbchar, pbblob, and pbstring. The second version is used to create a bounded array of strings.

Methods

PBBoundedArrayCreator has two methods:

- GetArray
- SetAt

GetArray

Obtains an array that has been created.

Syntax

GetArray()

Return value

pbarray

Examples

This example sets up an array, reads in values, and then obtains the values in the array:

```cpp
LPCTSTR *ostr_a;
char **sp;
int i;
pbarray out_array;
arrayBounds* bounds;
pbuint dim1, dim2, current_dim;
pblong itemcount1, itemcount2;
PBXRESULT ret;
PBArrayInfo* ai;
pbstring *iarg, *oarg;
typedef PBBoundedArrayCreator<pbvalue_string>
    BoundedStringArrayCreator;
in >> dim1;
// allocate memory for pointer bounds
bounds = (arrayBounds*)malloc(dim1*sizeof
        (PBArrayInfo::ArrayBound));
bounds = new arrayBounds[dim1];
// read in lowerbound and upperbound for each dimension
// and calculate the array item count
itemcount1 = 1;
for (i=0; i<dim1; i++)
```
{ 
  in >> bounds[i].lowerBound >> bounds[i].upperBound;
  itemcount1 = itemcount1*
    (bounds[i].upperBound - bounds[i].lowerBound +1);
}
sp = new char*[itemcount1];
ostr_a = new LPCTSTR[itemcount1];
iarg = new pbstring[itemcount1];
// Read in array items
for (i=0; i<itemcount1; i++)
{
  sp[i] = new char[20];
in >> sp[i];
iarg[i]= session->NewString(sp[i]);
}
// create bounded simple array and set iarg[i] to it
{
  BoundedStringArrayCreator ac(session, dim1, bounds);
current_dim = 1;
  BoundedArrayItem<pbstring, pbvalue_string, 
    BoundedStringArrayCreator>::f_set_arrayitem
    (session, ac, dim1, bounds, iarg, current_dim);
  BoundedArrayItem<pbstring, pbvalue_string,
    BoundedStringArrayCreator>::array_itemcount = 0;
  out_array = ac.GetArray();
}

See also SetAt

SetAt

Description
Sets a value or string to the array item at the specified dimension.

Syntax
For arrays of a specified ValueType:

SetAt(pblong dim[], ValueType v)

For string arrays:

SetAt(pblong dim[], LPCTSTR string)
SetAt(pblong dim[], pbstring string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>The dimension of the array item to be set</td>
</tr>
<tr>
<td>v</td>
<td>A ValueType defined in pbtraits.h</td>
</tr>
<tr>
<td>string</td>
<td>A string of type pbstring or LPCTSTR</td>
</tr>
</tbody>
</table>
**PBBoundedArrayCreator** template class

### Return value
None.

### Examples
This example shows the use of SetAt in arrays of a type specified by a ValueType:

```cpp
// arguments:
// ac: class object of PBBoundedArrayCreator or
// PBBoundedObjectArrayCreator to set items into
// dimensions: array dimension, can be 1,2,3,...,n
// bounds: upper and lower bound for each dimension
// iarg: T type array to store the data value set
// into array creator ac
// current_dim: remember which dimension is looped into

template <typename T, pbvalue_type I, class C>
void BoundedArrayItem<T, I, C>::f_set_arrayitem
(IPB_Session* session, C& ac, pblong dimensions,
arrayBounds* bounds, T* iarg, int current_dim)
{
int i;
if (current_dim > dimensions)
    return;
for(i= bounds[current_dim-1].lowerBound;
    i<= bounds[current_dim-1].upperBound; i++)
{
if (current_dim == dimensions)
{
dim[current_dim-1]= i;
ac.SetAt(dim, iarg[array_itemcount]);
array_itemcount++;
}
else
{
dim[current_dim-1]= i;
BoundedArrayItem<T, I, C>::f_set_arrayitem
    (session, ac, dimensions, bounds, iarg,
     current_dim+1);
}
}
```

See also
GetArray
**PBBoundedObjectArrayCreator class**

**Description**
The PBBoundedObjectArrayCreator class is used to create an object array.

**Methods**
PBBoundedObjectArrayCreator has two methods:
- GetArray
- SetAt

**GetArray**

**Description**
Obtains an array that has been created.

**Syntax**
GetArray()

**Return value**
pbarray.

**Examples**
This example sets the values in an array and then uses GetArray to obtain the array:

```c
PBBoundedObjectArrayCreator<pbvalue_string> ac(session);
for (i=0;i<itemcount1;i++)
{
    ac.SetAt(i+1,iarg[i]);
}
out_array = ac.GetArray();
```

**See also**
SetAt

**SetAt**

**Description**
Sets the array item at the specified dimension.

**Syntax**
For arrays of a specified ValueType:
```
SetAt(pblong dim[], ValueType v)
```

For string arrays:
```
SetAt(pblong dim[], LPCTSTR string)
SetAt(pblong dim[], pbstring string)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>The dimension of the array item to be set</td>
</tr>
<tr>
<td>v</td>
<td>A ValueType defined in pbtraits.h</td>
</tr>
<tr>
<td>string</td>
<td>A string of type pbstring or LPCTSTR</td>
</tr>
</tbody>
</table>
**PBObjectArrayAccessor class**

**Description**
The PBObjectArrayAccessor class is used to access the items in an object array.

**Methods**
PBObjectArrayAccessor has two methods:
- GetAt
- SetAt

---

**GetAt**

**Description**
Obtains the array item at the specified dimension.

**Syntax**
GetAt(pblong *dim[])

**Return value**
pbobject.

**Argument** | **Description**
--- | ---
*dim* | The dimension of the array item to be set

**Examples**
This example shows the use of GetAt in an object array:

```c
PBObjectArrayAccessor aa(session, *array_val);
for (i=0; i<itemcount2; i++)
{
    dim[0] = i+1;
    oarg = aa.GetAt(dim);
    cls = session->GetClass(oarg);
    if( cls == NULL )
        return;
    fid = session->GetFieldID(cls, "text");
    if ( fid == 0xffff)
        return;
    fid_pv = session->GetFieldAddress(oarg,fid);
    mystr = fid_pv->GetString();
    ostr_a[i] = session->GetString(mystr);
}
```

**See also**
SetAt
SetAt

Sets the array item at the specified dimension.

**Syntax**

```
SetAt(pblong dim[], pbobject obj)
```

**Argument** | **Description**
---|---
`dim` | The dimension of the array item to be set
`obj` | A valid object handle

**Return value**

None.

**Examples**

This example shows the use of SetAt in an object array:

```c
PBObjectArrayAccessor aa(session,*array_val);
for (i=0;i<itemcount1;i++)
{
    cls = session->FindClass(group,sp[i]);
    if( cls == NULL )
        return;
    iarg = session->NewObject(cls);
    session->ReferenceObject(iarg);
    dim[0] = i+1;
    aa.SetAt(dim, iarg);
    fid = session->GetFieldID(cls, "text");
    if ( fid == 0xffff )
        return;
    fid_pv = session->GetFieldAddress(iarg, fid);
    mystr = fid_pv->GetString();
    istr_a[i] = session->GetString(mystr);
}
```

**See also**

GetAt
PBUnboundedArrayCreator template class

Description
There are two versions of the PBUnboundedArrayCreator template class. The first version is used to create an unbounded array of a standard type. The standard types are defined as ValueTypes in pbtraits.h and are pbint, pbbyte, pbuint, pblong, pbulong, pbulonglong, pbboolean, pbreal, pdbdouble, pbdate, pbdate, pbtime, pbdatetime, pbschar, pbblob, and pbstring. The second version is used to create an unbounded array of strings.

Methods
PBUnboundedObjectArrayCreator has two methods:
  GetArray
  SetAt

GetArray
Description
Obtains an array that has been created.

Syntax
GetArray()

Return value
pbarray.

Examples
This example sets the values in an array and then uses GetArray to obtain the array:

    PBUnboundedArrayCreator<pbvalue_string> ac(session);
    for (i=0; i<itemcount1; i++)
    {
        ac.SetAt(i+1,iarg[i]);
    }
    out_array = ac.GetArray();

See also
SetAt
**SetAt**

**Description**
Sets the array item at the specified position.

**Syntax**
For arrays of a specified ValueType:

```
SetAt(pblong pos, ValueType v)
```

For string arrays:

```
SetAt(pblong pos, LPCTSTR string)
SetAt(pblong pos, pbstring string)
```

**Argument Description**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>A pblong identifying a position in the array</td>
</tr>
<tr>
<td>v</td>
<td>A ValueType defined in <code>pbtraits.h</code></td>
</tr>
<tr>
<td>string</td>
<td>A string of type pbstring or LPCTSTR</td>
</tr>
</tbody>
</table>

**Return value**
None.

**Examples**
This example shows the use of SetAt in arrays of a type specified by a ValueType:

```cpp
PBUnboundedArrayCreator<I> ac(session);
in >> iarg[i];
for (i=0; i<itemcount1; i++)
{
    ac.SetAt(i+1, iarg[i]);
}
out_array = ac.GetArray();
```

**See also**
GetArray
**PBUnboundedObjectArrayCreator class**

**Description**

The PBUnboundedObjectArrayCreator class is used to create an object array.

**Methods**

PBUnboundedObjectArrayCreator has two methods:

- GetArray
- SetAt

**GetArray**

**Description**

Obtains an array that has been created.

**Syntax**

GetArray( )

**Return value**

pbarray.

**See also**

SetAt

**SetAt**

**Description**

Sets the array item at the specified dimension.

**Syntax**

For arrays of a specified ValueType:

SetAt( pblong pos, ValueType v )

For string arrays:

SetAt( pblong pos, LPCTSTR string )

SetAt( pblong pos, pbstring string )

**Argument** | **Description**
--- | ---
pos | A pblong identifying a position in the array
v | A ValueType defined in pbtraits.h
string | A string of type pbstring or LPCTSTR

**Return value**

None.

**See also**

GetArray
Exported methods

The following table lists methods that must be implemented in the PowerBuilder extension module when the conditions shown in the table apply. The methods are described after the table. The PBX_GetVersion method is used by PowerBuilder to determine whether the compiler macro UNICODE or _UNICODE has been set. It is for internal use only.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBX_CreateNonVisualObject</td>
<td>When the extension contains nonvisual native classes</td>
</tr>
<tr>
<td>PBX_CreateVisualObject</td>
<td>When the extension contains visual native classes</td>
</tr>
<tr>
<td>PBX_DrawVisualObject</td>
<td>When you want to be able to draw a visual representation of the visual object in the PowerBuilder development environment</td>
</tr>
<tr>
<td>PBX_GetDescription</td>
<td>In all extensions</td>
</tr>
<tr>
<td>PBX_InvokeGlobalFunction</td>
<td>When the extension contains global functions</td>
</tr>
<tr>
<td>PBX_Notify</td>
<td>When you need to initialize and uninitialize a session</td>
</tr>
</tbody>
</table>

PBX_CreateNonVisualObject

Description

Creates a new instance of a nonvisual PowerBuilder extension object.

Syntax

PBX_CreateNonVisualObject(IPB_Session* pbsession, pbobject pobj, LPCTSTR xtraName, IPBX_NonVisualObject **obj);

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbsession</td>
<td>This IPB session</td>
</tr>
<tr>
<td>pobj</td>
<td>The name of a pbobject corresponding to the PowerBuilder extension object to be created</td>
</tr>
<tr>
<td>xtraname</td>
<td>The name of the PowerBuilder native class in lowercase</td>
</tr>
<tr>
<td>obj</td>
<td>The PowerBuilder extension object to be created</td>
</tr>
</tbody>
</table>

Return value

PBXRESULT. PBX_OK for success.
Examples

In this example, the extension contains several classes. The object created depends on the string value of the class name passed in.

```
PBXEXPORT PBXRESULT PBXCALL PBX_CreateNonVisualObject
{
    IPB_Session* pbsession,
    pbobject pbobj,
    LPCTSTR xtraName,
    IPBX_NonVisualObject **obj
}
{
    PBXRESULT result = PBX_OK;

    string cn(className);
    if (cn.compare("class_a") == 0)
    {
        *obj = new class_a(pbobj);
    }
    else if (cn.compare("class_b") == 0)
    {
        *obj = new class_b(pbobj);
    }
    else if (cn.compare("class_c") == 0)
    {
        *obj = new class_b(pbobj);
    }
    else
    {
        *obj = NULL;
        result = PBX_E_NO_SUCH_CLASS;
    }

    return PBX_OK;
};
```

Usage

You must implement this method in every PowerBuilder extension module that contains nonvisual classes. When you use the CREATE statement in PowerScript to create a new PowerBuilder extension object, the PBVM calls this method.

See also

PBX_GetDescription
PBX_CreateVisualObject

Description
Creates a new instance of a visual PowerBuilder extension object.

Syntax
PBX_CreateVisualObject(IPB_Session* pbsession, pbobject pbobj, LPCTSTR xtraName, IPBX_NonVisualObject **obj);

Return value
PBXRESULT. PBX_OK for success.

Examples
In this example the extension contains several classes. The object created depends on the string value of the class name passed in.

PBXEXPORT PBXRESULT PBXCALL PBX_CreateVisualObject
{
    IPB_Session* pbsession,
    pbobject pbobj,
    LPCTSTR className,
    IPBX_VisualObject **obj
}
{
    PBXRESULT result = PBX_OK;

    string cn(className);
    if (cn.compare("visualext") == 0)
    {
        *obj = new CVisualExt(pbsession, pbobj);
    }
    else
    {
        *obj = NULL;
        result = PBX_FAIL;
    }
    return PBX_OK;
};

Usage
You must implement this method in every PowerBuilder extension module that contains visual classes. When you use a visual extension in a PowerBuilder application, the PBVM calls this method.

See also
PBX_GetDescription
Exported methods

**PBX_DrawVisualObject**

**Description**

Draws a visual object in the PowerBuilder development environment.

**Syntax**

```c
PBX_DrawVisualObject(HDC hDC, LPCTSTR className, const PBX_DrawItemStruct& property);
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDC</td>
<td>A handle to the device context of the object</td>
</tr>
<tr>
<td>className</td>
<td>The name of the visual extension object to be drawn</td>
</tr>
<tr>
<td>property</td>
<td>A PBX_DrawItemStruct structure specifying the display properties of the object</td>
</tr>
</tbody>
</table>

**Return value**

PBXRESULT. The return value of this function is currently ignored.

**Examples**

This is an extension of a sample that is available on the PowerBuilder CodeXchange Web site at http://powerbuilder.codexchange.sybase.com. It draws a representation of a light-emitting diode (LED) and uses Microsoft Foundation Classes (MFC):

```c
PBXEXPORT PBXRESULT PBXCALL PBX_DrawVisualObject
{
    HDC hDC,
    LPCTSTR xstraName,
    const PBX_DrawItemStruct& property
}
{
    // If this PBX is dynamically linked against the MFC DLLs, any functions exported from this PBX that
    // call into MFC must have the AFX_MANAGE_STATE macro added at the very beginning of the function.
    AFX_MANAGE_STATE( AfxGetStaticModuleState() );

    // Variables to hold the Led control and a pointer
    // to Device Context
    CLed *myLed;
    CDC* pDC;

    // The name must not contain uppercase letters
    if ( strcmp( xstraName, "u_cpp_led" ) == 0 )
    {
        CRect rc( property.x, property.y, property.x + property.width, property.y + property.height );

        //Create a new LED
        myLed = new CLed();
```
// Get the handle from the hDC
pDC = CDC::FromHandle(hDC);
CWnd* pWnd = pDC->GetWindow();

// Create the window
myLed->Create(NULL, WS_CHILD | WS_VISIBLE |
    SS_BITMAP, rc, pWnd);

// Function that handles the background
// rendering of the control
myLed->OnEraseBkgndIDE(pDC);

// Draw the LED in default mode (red, on, round)
myLed->DrawLed(pDC,0,0,0);
myLed->SetLed(0,0,0);

//done
delete myLed;

return PBX_OK;

Usage
In a visual extension, export this function if you want the visual control to be
drawn in the development environment. If you do not export the function, you
need to run the application to see the appearance of the visual control.

See also
PBX_CreateVisualObject
PBX_DrawItemStruct structure
Exported methods

**PBX_GetDescription**

Description: Passes a description of all the classes and methods in the PowerBuilder extension module to PowerBuilder.

Syntax: `PBX_GetDescription()`

Return value: LPCTSTR containing the description of the module.

Examples: The following extension module contains three classes:

```c
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[] = {
        "class class_a from nonvisualobject\n"
        "function long meth1(string classpath)\n"
        "function string meth2()\n"
        "end class\n"

        "class class_b from nonvisualobject\n"
        "subroutine sbrt1()\n"
        "subroutine sbrt2()\n"
        "function long func1()\n"
        "end class\n"

        "class class_c from nonvisualobject\n"
        "end class\n"
    };

    return desc;
}
```

The following module contains a visual class that has two subroutines (functions that do not return values), two events that require that Windows messages be captured in the extension (onclick and ondoubleclick), and one event that maps a Windows message directly to a PowerBuilder event (testmouse). The module also contains two global functions, funcA and funcB.

```c
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[] = {
        "class visualext from userobject\n"
        "event int onclick()\n"
        "event int ondoubleclick()\n"
        "subroutine setcolor(int r, int g, int b)\n"
        "subroutine settext(string txt)\n"
        "event testmouse pbm_mousedown\n"
        "end class\n"
    };

    return desc;
}
```
"globalfunctions
"function int funca(int a, int b)
"function int funcb(int a, int b)
"end globalfunctions

};

return desc;
}

Usage

You must implement this method in every PowerBuilder extension module. The method is exported from the PowerBuilder extension module and is used by PowerBuilder to display the prototype of each class, function, and event in the module.

The syntax of the description follows:

Multiple instances
A syntax element with an asterisk indicates that multiple instances of that element can appear in a description. For example, \([Desc]*\) indicates that one description can contain multiple classes, global functions, and forward declarations.

\[
\text{Desc} := \begin{align*}
\text{class_desc} & | \text{globalfunc_desc} | \text{forward_desc} | [\text{Desc}]^* \\
\text{class_desc} & := \begin{align*}
\text{class} & \text{ className} \text{ from} \text{ parentClass} \text{ newline} \\
& [\text{methods_desc}]^* \text{ end class newline} \\
\text{globalfunc_desc} & := \text{globalfunctions newline [func_desc]* end globalfunctions} \\
\text{forward_desc} & := \text{forward newline [forwardtype_desc]* end forward} \\
\text{forwardtype_desc} & := \text{class className from parentClass newline} \\
\text{className} & := \text{a PowerBuilder token (cannot duplicate an existing group name)} \\
\text{parentClass} & := \text{any class inherited from NonVisualObject or UserObject} \\
\text{newline} & := \text{a newline character} \\
\text{methods_desc} & := \text{method_desc [methods_desc]*} \\
\text{method_desc} & := \text{func_desc | sub_desc | event_desc}
\end{align*}
\]
Exported methods

\[
\begin{align*}
\text{func_desc} & ::= \\
& \quad \text{function } \text{returnType} \text{Name}(\text{args_desc}) \text{ newline} \\
\text{returnType} & ::= \\
& \quad \text{pbType} \\
\text{pbType} & ::= \\
& \quad \text{any PowerBuilder type | previous declared PNI class} \\
\text{Name} & ::= \\
& \quad \text{a PowerBuilder token} \\
\text{args_desc} & ::= \\
& \quad \text{None | } \text{arg_desc}, [\text{args_desc}]^* \\
\text{arg_desc} & ::= \\
& \quad [\text{ref | readonly }] \text{pbType argName [array_desc]} \\
\text{argName} & ::= \\
& \quad \text{a PowerBuilder token} \\
\text{array_desc} & ::= \\
& \quad \text{array declaration of PowerBuilder} \\
\text{sub_desc} & ::= \\
& \quad \text{subroutine } \text{Name}(\text{args_desc}) \text{ newline} \\
\text{event_desc} & ::= \\
& \quad \text{event } \text{returnType} \text{Name}(\text{args_desc}) \text{ newline} \\
& \quad \text{event } \text{Name} \text{pbevent_token newline} \\
\text{pbevent_token} & ::= \\
& \quad \text{string} \\
\end{align*}
\]

This syntax for event_desc allows you to map a Windows message directly to a PowerBuilder event:

\[
\text{event } \text{Name} \text{pbevent_token newline}
\]

For more information, see “Event processing in visual extensions” on page 34.

See also
PBX_CreateNonVisualObject
PBX_CreateVisualObject
PBX_InvokeGlobalFunction
PBX_InvokeGlobalFunction

Description
Contains the implementation of one or more global functions used in the PowerBuilder extension module.

Syntax
PBX_InvokeGlobalFunction(IPB_Session* pbsession, LPCTSTR functionName, PBCallInfo* ci);

Argument | Description
--- | ---
pbsession | This IPB session
functionName | The name of the global function
ci | A pointer to a preallocated PBCallInfo structure containing the parameters and return value setting for the function

Return value
PBXRESULT. PBX_OK for success.

Examples
This PBX_GetDescription call declares three global functions: bitAnd, bitOr, and bitXor:

```
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[] = {
        "globalfunctions\n"
        "function int bitAnd(int a, int b)\n"
        "function int bitOr(int a, int b)\n"
        "function int bitXor(int a, int b)\n"
        "end globalfunctions\n"
    };

    return desc;
}
```

The PBX_InvokeGlobalFunction call contains the implementation of the functions:

```
PBXEXPORT PBXRESULT PBXCALL PBX_InvokeGlobalFunction
{
    IPB_Session* pbsession,
    LPCTSTR functionName,
    PBCallInfo* ci
}
```

```
PBXRESULT pbrResult = PBX_OK;

int arg1 = ci->pArgs->GetAt(0)->GetInt();
int arg2 = ci->pArgs->GetAt(1)->GetInt();
```
if (stricmp(functionName, "bitand") == 0)
{
    ci->returnValue->SetInt(arg1 & arg2);
} else if (stricmp(functionName, "bitor") == 0)
{
    ci->returnValue->SetInt(arg1 | arg2);
} else if (stricmp(functionName, "bitxor") == 0)
{
    ci->returnValue->SetInt(arg1 ^ arg2);
} else
{
    return PBX_FAIL;
}

return pbrResult;

Usage

Use this function in a PowerBuilder native class that uses global functions. The function is exported from the PowerBuilder extension module and is used to identify global functions included in the module. Like global functions in PowerScript, global functions in PowerBuilder extensions cannot be overloaded.

See also

PBX_GetDescription
CHAPTER 7  PBINI Interfaces, Structures, and Methods

**PBX_Notify**

**Description**
Used to initialize and uninitialize a session.

**Syntax**

```
PBXEXPORT PBXRESULT PBXCALL PBX_Notify(IPB_Session* pbsession, pbint reasonForCall)
```

**Return value**

PBXRESULT

**Examples**

This sample shows code that exports PBX_Notify and displays a message box after the PBX is loaded and before it is unloaded:

```
PBXEXPORT PBXRESULT PBXCALL PBX_Notify
{
    IPB_Session* pbsession,
    pbint reasonForCall
}
{
    switch(reasonForCall)
    {
    case kAfterDllLoaded:
        MessageBox(NULL, "After PBX loading", ",",
        MB_OK);
        break;
    case kBeforeDllUnloaded:
        MessageBox(NULL, "Before PBX unloading", ",",
        MB_OK);
        break;
    }
    return PBX_OK;
}
```

**Usage**

If PBX_NOTIFY is exported, the PBVM calls PBX_Notify immediately after an extension PBX is loaded and just before the PBX is unloaded. You can use this function to initialize and uninitialize a session. For example, you could create a session manager object, and store it in the IPB session using the SetProp function. Later, you could use GetProp to obtain the session object.
Method exported by PowerBuilder VM

Description
This method is exported by the PowerBuilder VM:

PB_GetVM

PB_GetVM

Description
Passes the IPB_VM interface to the user.

Syntax
PB_GetVM (IPB_VM** vm)

Examples
This example loads the PowerBuilder VM and calls the f_getrowcount function on the nvo_dw custom class user object:

```cpp
#include <pbext.h>
#include <iostream.h>
typedef PBXEXPORT PBXRESULT (*P_PB_GetVM)(IPB_VM** vm);

class LibraryLoader
{
public:
    LibraryLoader(LPCSTR libname)
    {
        d_hinst = LoadLibrary(libname);
    }

    ~LibraryLoader()
    {
        FreeLibrary(d_hinst);
    }

    operator HINSTANCE()
    {
        return d_hinst;
    }

private:
    HINSTANCE d_hinst;
};

int main()
{
    int int_rowcount;
    PBXRESULT ret;
    LibraryLoader loader("pbvm125.dll");
```
if ((HINSTANCE)loader == NULL) return 0;

P_PB_GetVM getvm = (P_PB_GetVM)
    GetProcAddress((HINSTANCE)loader, "PB_GetVM");
if (getvm == NULL) return 0;

IPB_VM* vm = NULL;
getvm(&vm);
if (vm == NULL) return 0;

static const char *liblist[] =
{
    "load_pbvm.pbl"
};

IPB_Session* session = NULL;
ret = vm->CreateSession
    ("load_pbvm", liblist, 1, &session);
if (ret != PBX_OK)
{
    cout << " Create session failure!" << endl;
    return 0;
}
return 1;

Usage
To load the PowerBuilder VM and run a PowerBuilder application in a third-party server or application, you first create an IPB_VM object using the PB_GetVM method. Then, create an IPB_Session object within IPB_VM, using the application’s name and library list as arguments.

See also
CreateSession
Method exported by PowerBuilder VM
CHAPTER 8

PBNI Tool Reference

About this chapter

This chapter describes two tools provided with the PBNI SDK:

- The pbsig125 tool gets the internal signature of a PowerBuilder function from a PBL name.
- The pbx2pbd125 tool generates a PBD from a PBX.

When you install PowerBuilder, these tools are installed in the SDK subdirectory of your PowerBuilder 12.5 directory and in Shared\PowerBuilder.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbsig125</td>
<td>255</td>
</tr>
<tr>
<td>pbx2pbd125</td>
<td>259</td>
</tr>
</tbody>
</table>

pbsig125

Description

The PowerBuilder function signature is the internal signature of a PowerBuilder function that is used to identify polymorphism functions in a class. The pbsig125 tool obtains these function signatures from a PBL.

Inherited functions

You can also obtain a signature by selecting the function in the System Tree or Browser and selecting Properties from its pop-up menu. The pbsig125 tool does not report the signature of functions that are inherited from an ancestor object unless they are extended in the descendant. For such functions, you must use the Properties dialog box to obtain the signature. The Properties dialog box in the Browser also allows you to obtain the signature of PowerBuilder system functions.

Syntax

pbsig125 pbl_name
This command extracts function signatures from one of the PBLs in the Code Examples sample application:

```
pbsig125 pbexamw1.pbl
```

Here is some of the output from the previous command:

```powershell
PB Object Name: w_date_sort
   public subroutine of_sort (string as_Column,
               string as_Order)
      /* QSS */

PB Object Name: w_date_window
   public function boolean of_is_leap_year
      (integer ai_year)
      /* BI */

   public subroutine of_days ()
      /* Q */

PB Object Name: w_dde_server
   public subroutine check_hotlink (checkbox status,
               string data, string item)
      /* QCcheckbox.SS */

PB Object Name: w_dir_tree
   public function integer
      wf_collapse_rows (datawindow
               adw_datawindow, long al_startrow)
      /* ICdatawindow.L */

   public function long of_recurse_dir_list (string
               as_path, long al_parent)
      /* LSL */

   public function string of_build_dw_tree
      (long al_handle)
      /* SL */
```

The following example illustrates the use of a letter code to represent a PowerBuilder system class or a custom class. Consider this function:

```powershell
function integer of_get_all_sales_orders (Ref
               s_sales_order astr_order[], date adt_date, integer
               ai_direction)
```

For this function, the `pbsig125` tool returns the following string. The first argument is an unbounded array of type `s_sales_order` and is passed by reference:

```powershell
/* IRCs_sales_order.[]YI */
```
Usage

The pbsig125 tool generates a string that represents the declaration and signature of all the functions and events in the PBL, including argument types, return types, and passing style. Each function and event is followed by a commented string. You pass the commented string, for example, QSS in the first comment in the previous example, as the last argument to the GetMethodID method.

For example, the following output indicates that the function returns an integer and has a single integer argument passed by reference:

```c
/* IRI */
```

**PowerBuilder arrays**  
PowerBuilder arrays are indicated with a pair of square brackets [[]] as a suffix. For bounded arrays, the brackets enclose the bounds.

```c
/* IRCdatastore.RS[]SS */
```

**PowerBuilder system or custom class**  
Additional letter codes represent a PowerBuilder system class or a custom class. The letter C followed by the name of a PowerBuilder object or enumerated class and a period (Cname.) represents an argument or return value of that type.

The following table shows how the output from pbsig125 maps to datatypes and other entities.

<table>
<thead>
<tr>
<th>Output</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>array</td>
</tr>
<tr>
<td>A</td>
<td>any</td>
</tr>
<tr>
<td>B</td>
<td>boolean</td>
</tr>
<tr>
<td>C</td>
<td>class</td>
</tr>
<tr>
<td>D</td>
<td>double</td>
</tr>
<tr>
<td>E</td>
<td>byte</td>
</tr>
<tr>
<td>F</td>
<td>real</td>
</tr>
<tr>
<td>G</td>
<td>basictype</td>
</tr>
<tr>
<td>H</td>
<td>character</td>
</tr>
<tr>
<td>I</td>
<td>integer</td>
</tr>
<tr>
<td>J</td>
<td>cursor</td>
</tr>
<tr>
<td>K</td>
<td>longlong</td>
</tr>
<tr>
<td>L</td>
<td>long</td>
</tr>
<tr>
<td>M</td>
<td>decimal</td>
</tr>
<tr>
<td>N</td>
<td>unsigned integer (uint)</td>
</tr>
<tr>
<td>O</td>
<td>blob</td>
</tr>
</tbody>
</table>
The passing style is indicated by a prefix on the type.

**Table 8-2: Passing style and varargs representation in pbsig125 output**

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Pass by value</td>
</tr>
<tr>
<td>R</td>
<td>Pass by reference</td>
</tr>
<tr>
<td>X</td>
<td>Pass as read only</td>
</tr>
<tr>
<td>V</td>
<td>Variable arguments (varargs)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>dbproc</td>
</tr>
<tr>
<td>Q</td>
<td>No type (subroutine)</td>
</tr>
<tr>
<td>S</td>
<td>string</td>
</tr>
<tr>
<td>T</td>
<td>time</td>
</tr>
<tr>
<td>U</td>
<td>unsigned long (ulong)</td>
</tr>
<tr>
<td>W</td>
<td>datetime</td>
</tr>
<tr>
<td>Y</td>
<td>date</td>
</tr>
<tr>
<td>Z</td>
<td>objhandle</td>
</tr>
</tbody>
</table>
pbx2pbd125

Description
The pbx2pbd125 tool generates a PowerBuilder dynamic library (PBD) file from a PowerBuilder extension PBX. The generated PBD can be added to the library list of any PowerBuilder application target that will use the objects and methods in the PowerBuilder extension.

Syntax
```
px2pbd125 [+] des.pbd src1.pbx [ src2.pbx src3.pbx ...srcn.pbx ]
```

Examples
This example generates a new PBD test.pbd from test.pbx. The input and output files are in the current directory:
```
px2pbd125 test.pbd test.pbx
```
This example appends generated information from C:\myproject\src.pbx to C:\mypbds\des.pbd. (If des.pbd does not exist, it is created.)
```
px2pbd125 + C:\mypbds\des.pbd C:\myproject\src.pbx
```
This example generates a new PBD D:\pbds\test.pbd from all the PBX files in the C:\myproject directory:
```
px2pbd125 D:\pbds\test.pbd C:\myproject\*.pbx
```
This example generates PBD information from all the PBX files in the C:\temp and D:\temp directories and appends the information to the existing generated PBD file D:\pbds\test.pbd:
```
px2pbd125 + D:\pbds\test.pbd c:\temp\*.pbx
d:\temp\*.pbx
```

Usage
You can import an extension into a PowerBuilder library using the Import PB Extension pop-up menu item for the library in the PowerBuilder System Tree. Prior to PowerBuilder 11.5, you had to use the pbx2pbdnnn tool to create a PBD file from a PBX file, then add the PBD to the library list of your PowerScript target. The tool is still available in this release.

You can include multiple PBXs in a single PBD file. If you want to add additional PBXs to an existing PBD, use the plus (+) sign before the name of the PBD.

The pbx2pbd125 tool is installed in the system PATH in the Shared\PowerBuilder directory so you can invoke it in the directory where the PBXs reside.

If you specify an absolute path for the PBX file when you generate the PBD, the PowerBuilder application searches for the PBX only in the specified path.

If you do not specify the path for the PBX file, the PowerBuilder application searches the system path for the PBX.
This appendix describes wizards provided for Microsoft Visual Studio.
APPENDIX A

Using the Visual Studio Wizards

About this appendix

If you use Visual Studio .NET 2002 or 2003 or Visual Studio 2005, you can use a wizard to create a PBNI extension project. The wizard creates a project with .cpp and .h files that contain required code as well as template code to help you get started.

Check for wizard updates in the PBNI section of the PowerBuilder CodeXchange Web site at http://powerbuilder.codeXchange.sybase.com/.

Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where the wizards are installed</td>
<td>263</td>
</tr>
<tr>
<td>Generating a PBNI project</td>
<td>265</td>
</tr>
<tr>
<td>Setting project options</td>
<td>266</td>
</tr>
<tr>
<td>Building and using the PBX</td>
<td>266</td>
</tr>
</tbody>
</table>

Where the wizards are installed

When you install PowerBuilder, the setup program installs four directories into the PowerBuilder 12.5\SDK\PBNI\wizards directory:

- VCPProjects 7.0
- VCPProjects 7.1
- VCPProjects 8.0
- VCWizards

If Microsoft Visual Studio is already installed on your computer, the setup program also installs the appropriate files into your Visual Studio installation.
Where the wizards are installed

<table>
<thead>
<tr>
<th>Visual Studio version</th>
<th>Files copied</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Studio 2005</td>
<td>VCPProjects 8.0</td>
<td>..\Microsoft Visual Studio 8\VC\VCPProjects</td>
</tr>
<tr>
<td></td>
<td>VCWizards</td>
<td>..\Microsoft Visual Studio 8\VC\VCWizards</td>
</tr>
<tr>
<td>Visual Studio .NET 2003</td>
<td>VCPProjects 7.1</td>
<td>..\Microsoft Visual Studio .NET 2003\VC\VCPProjects</td>
</tr>
<tr>
<td></td>
<td>VCWizards</td>
<td>..\Microsoft Visual Studio .NET 2003\VC\VCWizards</td>
</tr>
<tr>
<td>Visual Studio .NET 2002</td>
<td>VCPProjects 7.0</td>
<td>..\Microsoft Visual Studio .NET 2002\VC\VCPProjects</td>
</tr>
<tr>
<td></td>
<td>VCWizards</td>
<td>..\Microsoft Visual Studio .NET 2002\VC\VCWizards</td>
</tr>
</tbody>
</table>

If Visual Studio is not already installed when you install PowerBuilder, see the install.txt file in the PBNI\wizards\VCWizards\PBNIWizard directory for how to install the wizard later.

To check whether the wizard is installed in Visual Studio, select File>New>Project from the menu bar, select Visual C++ Projects, and scroll the Templates pane to see the PBNI wizard.
Generating a PBNI project

The PBNI Application Wizard lets you choose whether to create a visual or nonvisual extension, whether to include support for Unicode and global functions, and whether to generate a header file.

❖ To create a new PBNI project:

1 Start Visual Studio, select File > New > Project, select Visual C++ Projects, and scroll the Templates pane to see the PBNI wizard.

2 Select PBNI Extension DLL, enter a name and location for the project, and click OK.

3 Click Application Settings if you want to create a visual extension or change any other settings.

   The default is to create a nonvisual extension with Unicode support.

4 Click Finish.

   See the ReadMe.txt file created by the wizard for a description of the generated source and header files.
Setting project options

If the project does not build correctly, you might need to turn off precompiled headers in the project’s Property Pages dialog box and set the path for the PBNI include files.

❖ To set project options for PBNI library and include files:
1. In Visual Studio, select Tools>Options.
2. Select Projects and Solutions>VC++ Directories.
3. Select Include Files from the Show Directories For drop-down list and click the New icon. Then click the browse button, browse to the location of the PowerBuilder 12.5\SDK\PBNI\include directory, and click OK.
4. Click OK to close the Options dialog box.

By default, the project is compiled for Unicode character sets. You can change this setting in the wizard. If you want to change it to compile for ASCII (SBCS) character sets after you have created the project, you can remove the _UNICODE preprocessor option.

❖ To compile for ASCII character sets:
1. Select Project>ProjectName Properties.
2. Expand C/C++ and select Preprocessor.
3. Edit the Preprocessor Definitions to remove _UNICODE and UNICODE.

Building and using the PBX

When you have finished coding the project, build the project from the Build menu to create a DLL with the extension .pbx. By default, the extension is created in the Debug directory.

Then, you can import the PBX into a PBL in your PowerBuilder target and use it as described in “Using the extension” on page 27.
## Index

### A
- AcquireArrayItemValue function (IPB_Session) 102
- AcquireValue function (IPB_Session)
  - description 103
  - using 62
- AddArrayArgument function (IPB_Session) 104
- AddBlobArgument function (IPB_Session) 104
- AddBoolArgument function (IPB_Session) 104
- AddByteArgument function (IPB_Session) 104
- AddCharArgument function (IPB_Session) 104
- AddDateArgument function (IPB_Session) 104
- AddDateTimeArgument function (IPB_Session) 104
- AddDateTimeArgument function (IPB_Session) 104
- AddDecArgument function (IPB_Session) 104
- AddDoubleArgument function (IPB_Session) 104
- AddGlobalRef function (IPB_Session) 106
- AddIntArgument function (IPB_Session) 104
- AddLocalRef function (IPB_Session) 107
- AddLongArgument function (IPB_Session) 104
- AddLongLongArgument function (IPB_Session) 104
- AddObjectArgument function (IPB_Session) 104
- AddPBStringArgument function (IPB_Session) 104
- AddRealArgument function (IPB_Session) 104
- AddRef function (IPB_Session)
  - IPBX_Marshaler interface 92, 95
  - IPBX_UserObject interface 219
- AddStringArgument function (IPB_Session) 104
- AddTimeArgument function (IPB_Session) 104
- AddUintArgument function (IPB_Session) 104
- AddUlongArgument function (IPB_Session) 104

### C
- C++
  - calling PowerBuilder from 65
  - coding extensions 12
  - calling PowerScript from an extension 39
- ClearException function (IPB_Session) 107
- code samples, on Web site 3

### conventions xi
- CreateControl function (IPBX_VisualObject) 221
  - using 33
- CreateResultSet function (IPB_Session) 108
- CreateSession function (IPB_VM) 211

### D
- data
  - exchanging 55
  - saving values 61
- datatypes
  - for PowerBuilder data access 84
  - PowerBuilder to PBNI mapping 83
- debugging 43
- Destroy function (IPBX_Marshaler) 215
- Destroy function (IPBX_UserObject) 219

### E
- enumerated types
  - pbgroup 84
  - pbroutine 85
  - pbvalue 85
  - support for 64, 129
- event IDs, and triggering events 37
- event processing, in visual extensions 34
- examples
  - calling PowerBuilder functions 42
  - nonvisual extension 11
  - on the Web 3
- exception handling 43
- exchanging data with PowerBuilder 55
- exported methods 241
- exporting methods 24
- extensions
  - creating 17, 28
  - marshaler, about 6
Index

marshaler, creating 45
nonvisual, about 4
nonvisual, describing 19
nonvisual, example 11
using in PowerBuilder 26
visual, about 5
visual, creating 28
visual, creating instances 31
visual, event processing 34
visual, using 30

F
FindClass function (IPB_Session) 111
FindClassByClassID function (IPB_Session) 111
FindGroup function (IPB_Session) 112
FindMatchingFunction function (IPB_Session) 112
forward declarations 20
FreeCallInfo function (IPB_Session) 114

G
GetArray function (IPB_Value) 203
GetArray function (PBBoundedObjectArrayCreator class) 235
GetArray function (PBUnboundedObjectArrayCreator template class) 238
GetArray function (PBUnboundedObjectArrayCreator class) 240
GetArray method (PBBoundedArrayCreator template class) 232
GetArrayField function (IPB_Session) 116
GetArrayGlobalVar function (IPB_Session) 117
GetArrayInfo function (IPB_Session) 120
GetArrayItemType (IPB_Session) 121
GetArrayLength function (IPB_Session) 122
GetArraySharedVar function (IPB_Session) 118, 144
GetAt (PBArrayAccessor template class) 229
GetAt (PBOBJECTArrayAccessor template class) 236
GetAt function (IPB_Arguments) 90
GetBlob function (IPB_Session) 123
GetBlob function (IPB_Value) 203
GetBlobArrayItem function (IPB_Session) 114
GetBlobField function (IPB_Session) 116
GetBlobGlobalVar function (IPB_Session) 117
GetBlobSharedVar function (IPB_Session) 118, 144
GetBool function (IPB_Value) 203
GetBoolArrayItem function (IPB_Session) 114
GetBoolField function (IPB_Session) 116
GetBoolGlobalVar function (IPB_Session) 117
GetBoolSharedVar function (IPB_Session) 118, 144
GetByte function (IPB_Value) 203
GetByteArrayItem function (IPB_Session) 114
GetByteField function (IPB_Session) 116
GetByteGlobalVar function (IPB_Session) 117
GetByteSharedVar function (IPB_Session) 118
GetChar function (IPB_Value) 203
GetCharArrayItem function (IPB_Session) 114
GetCharField function (IPB_Session) 116
GetCharGlobalVar function (IPB_Session) 117
GetCharSharedVar function (IPB_Session) 118, 144
GetClass function (IPB_Session) 125, 204
GetClassName function (IPB_Session) 126
GetColumnCount function (IPB_ResultSetAccessor interface) 92
GetColumnMetaData function (IPB_ResultSetAccessor interface) 93
GetCount function (IPB_Arguments struct) 91
GetCurrGroup function (IPB_Session) 126
GetDate function (IPB_Value) 203
GetDateArrayItem function (IPB_Session) 114
GetDateField function (IPB_Session) 116
GetDateGlobalVar function (IPB_Session) 117
GetDateSharedVar function (IPB_Session) 118, 144
GetDateTime function (IPB_Value) 203
GetDateTimeArrayItem function (IPB_Session) 114
GetDateTimeField function (IPB_Session) 116
GetDateTimeGlobalVar function (IPB_Session) 117
GetDateTimeSharedVar function (IPB_Session) 118, 144
GetDateTimeString function (IPB_Session) 127
GetDateTimeTime function (IPB_Value) 203
GetDateTimeTimeArrayItem function (IPB_Session) 114
GetDateTimeTimeField function (IPB_Session) 116
GetDateTimeTimeGlobalVar function (IPB_Session) 117
GetDateTimeTimeSharedVar function (IPB_Session) 118, 144
GetDateTimeTimeString function (IPB_Session) 127
GetDec function (IPB_Value) 203
GetDecArrayItem function (IPB_Session) 114
GetDecField function (IPB_Session) 116
GetDecGlobalVar function (IPB_Session) 117
GetDecimalString function (IPB_Session) 128
GetDecSharedVar function (IPB_Session) 118, 144
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetDouble function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetDoubleField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetDoubleGlobalVar function (IPB_Session)</td>
<td>117</td>
</tr>
<tr>
<td>GetDoubleSharedVar function (IPB_Session)</td>
<td>118, 144</td>
</tr>
<tr>
<td>GetEnumItemName function (IPB_Session)</td>
<td>128</td>
</tr>
<tr>
<td>GetEnumItemValue function (IPB_Session)</td>
<td>129</td>
</tr>
<tr>
<td>GetEventID function (IPBX_VisualObject)</td>
<td>223</td>
</tr>
<tr>
<td>GetException function (IPB_Session)</td>
<td>130</td>
</tr>
<tr>
<td>GetFieldID function (IPB_Session)</td>
<td>130</td>
</tr>
<tr>
<td>GetFieldName function (IPB_Session)</td>
<td>131</td>
</tr>
<tr>
<td>GetFieldType function (IPB_Session)</td>
<td>132</td>
</tr>
<tr>
<td>GetGlobalVarID function (IPB_Session)</td>
<td>132</td>
</tr>
<tr>
<td>GetGlobalVarType function (IPB_Session)</td>
<td>133</td>
</tr>
<tr>
<td>GetInt function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetIntArrayItem function (IPB_Session)</td>
<td>114</td>
</tr>
<tr>
<td>GetIntField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetIntGlobalVar function (IPB_Session)</td>
<td>117</td>
</tr>
<tr>
<td>GetIntSharedVar function (IPB_Session)</td>
<td>118</td>
</tr>
<tr>
<td>GetItemData function (IPB_ResultSetAccessor interface)</td>
<td>94</td>
</tr>
<tr>
<td>GetLong function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetLongField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetLongGlobalVar function (IPB_Session)</td>
<td>117, 118, 144</td>
</tr>
<tr>
<td>GetLongLong function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetLongLongArrayItem function (IPB_Session)</td>
<td>114</td>
</tr>
<tr>
<td>GetLongLongField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetLongLongGlobalVar function (IPB_Session)</td>
<td>117</td>
</tr>
<tr>
<td>GetMethodID function (IPB_Session)</td>
<td>135</td>
</tr>
<tr>
<td>GetMethodID function (IPB_Session), about</td>
<td>39</td>
</tr>
<tr>
<td>GetMethodIDByEventID function (IPB_Session)</td>
<td>136</td>
</tr>
<tr>
<td>GetModuleHandle function (IPBX_Marshaler)</td>
<td>216</td>
</tr>
<tr>
<td>GetNativeInterface function (IPB_Session)</td>
<td>137</td>
</tr>
<tr>
<td>GetNumOfFields function (IPB_Session)</td>
<td>138</td>
</tr>
<tr>
<td>GetObject function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetObjectArrayItem function (IPB_Session)</td>
<td>114</td>
</tr>
<tr>
<td>GetObjectField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetObjectGlobalVar function (IPB_Session)</td>
<td>117</td>
</tr>
<tr>
<td>GetObjectSharedVar function (IPB_Session)</td>
<td>118, 144</td>
</tr>
<tr>
<td>GetPBAryArrayItem function (IPB_Session)</td>
<td>138</td>
</tr>
<tr>
<td>GetPBAnyField function (IPB_Session)</td>
<td>139</td>
</tr>
<tr>
<td>GetPBAnyGlobalVar function (IPB_Session)</td>
<td>141</td>
</tr>
<tr>
<td>GetPBAnySharedVar function (IPB_Session)</td>
<td>141</td>
</tr>
<tr>
<td>GetProp function (IPB_Session)</td>
<td>142</td>
</tr>
<tr>
<td>GetReal function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetRealArrayItem function (IPB_Session)</td>
<td>114</td>
</tr>
<tr>
<td>GetRealField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetRealGlobalVar function (IPB_Session)</td>
<td>117</td>
</tr>
<tr>
<td>GetRealSharedVar function (IPB_Session)</td>
<td>118, 144</td>
</tr>
<tr>
<td>GetResultSetAccessor function (IPB_Session)</td>
<td>142</td>
</tr>
<tr>
<td>GetRowCount function (IPB_ResultSetAccessor interface)</td>
<td>94</td>
</tr>
<tr>
<td>GetSharedVarID function (IPB_Session)</td>
<td>143</td>
</tr>
<tr>
<td>GetSharedVarType function (IPB_Session)</td>
<td>144</td>
</tr>
<tr>
<td>GetString function (IPB_Session)</td>
<td>145</td>
</tr>
<tr>
<td>GetString function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetStringArrayItem function (IPB_Session)</td>
<td>114</td>
</tr>
<tr>
<td>GetStringField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetStringGlobalVar function (IPB_Session)</td>
<td>117</td>
</tr>
<tr>
<td>GetStringLength function (IPB_Session)</td>
<td>146</td>
</tr>
<tr>
<td>GetStringSharedVar function (IPB_Session)</td>
<td>118</td>
</tr>
<tr>
<td>GetSuperClass function (IPB_Session)</td>
<td>146</td>
</tr>
<tr>
<td>GetSystemClass function (IPB_Session)</td>
<td>147</td>
</tr>
<tr>
<td>GetSystemGroup function (IPB_Session)</td>
<td>147</td>
</tr>
<tr>
<td>GetTime function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetTimeArrayItem function (IPB_Session)</td>
<td>114</td>
</tr>
<tr>
<td>GetTimeField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetTimeGlobalVar function (IPB_Session)</td>
<td>117</td>
</tr>
<tr>
<td>GetTimeString function (IPB_Session)</td>
<td>148</td>
</tr>
<tr>
<td>GetType function (IPB_Value)</td>
<td>204</td>
</tr>
<tr>
<td>GetUint function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetUintArrayItem function (IPB_Session)</td>
<td>114</td>
</tr>
<tr>
<td>GetUintField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetUintGlobalVar function (IPB_Session)</td>
<td>117</td>
</tr>
<tr>
<td>GetUintSharedVar function (IPB_Session)</td>
<td>118, 144</td>
</tr>
<tr>
<td>GetUlong function (IPB_Value)</td>
<td>203</td>
</tr>
<tr>
<td>GetUlongArrayItem function (IPB_Session)</td>
<td>114</td>
</tr>
<tr>
<td>GetUlongField function (IPB_Session)</td>
<td>116</td>
</tr>
<tr>
<td>GetUlongGlobalVar function (IPB_Session)</td>
<td>117</td>
</tr>
<tr>
<td>GetUlongSharedVar function (IPB_Session)</td>
<td>118, 144</td>
</tr>
<tr>
<td>GetWindowClassName function (IPBX_VisualObject)</td>
<td>225</td>
</tr>
</tbody>
</table>

Index

PBNI Programmers Guide and Reference 269

global functions 7
    declaring 21, 22
    describing 20
Index

exporting 20, 249
implementing 24

H
HasExceptionThrown function (IPB_Session) 148
helper classes 7, 9

I
InitCallInfo function (IPB_Session) 150
using 39
interfaces
  IPB_Arguments 90
  IPB_Session 96
  IPB_Value 202
  IPB_VM 211
  IPBX_Marshaler 215
  IPBX_NonVisualObject 218
  IPBX_UserObject 219
  IPBX_VisualObject 221
overview 7
Invoke function (IPBX_UserObject) 220
InvokeClassFunction function (IPB_Session) 151
InvokeObjectFunction function (IPB_Session) 152
InvokeRemoteMethod function (IPBX_Marshaler) 217
IPB_Arguments
  about 56
  description 90, 92, 95
  using 56
IPB_Session
  about 59
  interface description 96
  list of functions 96
  using 59
IPB_Value
  about 57
  description 202
  list of methods 202
  saving data 61
  using 57
IPB_VM
  about 6
  functions 211

IPBX_Marshaler interface 215
IPBX_NonVisualObject interface 218
IPBX_NonVisualUserObject interface 218
IPBX_UserObject interface 219
IPBX_VisualObject interface 221
IsArray function (IPB_Value) 205
IsArrayItemNull function (IPB_Session) 153
IsAutoInstantiate function (IPB_Session) 153
IsByRef function (IPB_Value) 205
IsEnum function (IPB_Value) 206
IsFieldArray function (IPB_Session) 153
IsFieldNull function (IPB_Session) 154
IsFieldObject function (IPB_Session) 155
IsGlobalVarArray function (IPB_Session) 155
IsGlobalVarNull function (IPB_Session) 156
IsGlobalVarObject function (IPB_Session) 157
IsNull function (IPB_Value) 206
IsNull function (PArrayAccessor template class) 229
IsObject function (IPB_Value) 207
IsSharedVarArray function (IPB_Session) 159
IsSharedVarNull function (IPB_Session) 159
IsSharedVarObject function (IPB_Session) 160

J
Java, calling from PowerBuilder 54
JNI
calling extensions 54
compared with PBNI 10
used with marshaler extension 6
using with PBNI 78

M
marshaler extensions
  about 6
  creating 45
  describing 47
  developing 46
  generating proxies 53
  implementing Creator class 48
  implementing marshaler class 51
overview 45
method ID
Index

and GetMethodID 39
getting 255

N
native class
declaring 21
describing 246
implementing 22
NewBlob function (IPB_Session) 160
NewBoundedObjectArray function (IPB_Session)
161
NewBoundedSimpleArray function (IPB_Session)
162
NewDate function (IPB_Session) 163
NewDateTime function (IPB_Session) 164
NewDecimal function (IPB_Session) 164
NewObject function (IPB_Session) 165
NewProxyObject function (IPB_Session) 166
NewString function (IPB_Session) 167
NewTime function (IPB_Session) 167
NewUnboundedObjectArray function (IPB_Session)
168
NewUnboundedSimpleArray function (IPB_Session)
169
nonvisual classes
creating instances 25
describing 19
using 27
nonvisual extensions
about 4
building 17
describing 19
example 11

P
passing values 55
PB_DateData structure 227
PB_DateTimeData structure 227
PB_GetVM function (exported from PBVM) 252
PB_TimeData structure 227
PBArrayAccessor
in pbarray.cpp 9
template class 229
PBArryInfo structure 226
PBArryInfoHolder 88
PBBoundedArrayCreator
in pbarray.cpp 9
PBBoundedArrayCreator template class 232
PBBoundedObjectArrayCreator
in pbarray.cpp 9
template class 235
PBCallInfo structure
reference 226
saving data 61
using 56
PBD
adding to library list 259, 260
generating 259
PBEventTrigger 88
in pbfuninv.cpp 9
pbevt.awx 263
PGBlobalFunctionInvoker 88
in pbfuninv.cpp 9
pbggroup enumerated types 84
PBNI
introduction 3
Software Development Kit (SDK) 8
PBObjectorArrayAccesso
class 236
in pbarray.cpp 9
PBObjector, in pbobject.cpp 9
PBObjectorFunctionInvoker 88
in pbfuninv.cpp 9
pbroutine enumerated types 85
pbsig110
datatype mapping 257
tool 255
PBUunboundedArrayCreator
in pbarray.cpp 9
template class 238
PBUunboundedObjectArrayCreator
in pbarray.cpp 9
template class 240
pbovalue enumerated types 85
PBVM
embedding in a C++ application 6
interaction with extension 8
loading 252

PNI NNI Progrmrms Guide and Reference 271
Index

PBX
  building 26
  importing 26
  using instead of DLL as extension file type 11
PBX_CreateNonVisualObject function 18, 241
PBX_CreateVisualObject function 243
PBX_DrawItemStruct structure 228
PBX_DrawVisualObject function 244
PBX_GetDescription function 18, 246
  using 18
PBX_GetVersion function 241
PBX_InvokeGlobalFunction function 249
PBX_Notify function 251
  using with SetProp 64
px2pbd tool 259
PBXResult error codes 86
PBXRuntimeError exception 43
PopLocalFrame function (IPB_Session) 170
PowerBuilder extensions
  building 11, 17, 26
  calling PowerScript from 39
  example 11
  marshaler extensions 6, 45
  nonvisual 4
  overview 4
  planning 18, 28
  using 27
PBXResult error codes 86
PBXRuntimeError exception 43
PopLocalFrame function (IPB_Session) 170
PushLocalFrame function (IPB_Session) 172

R
ReferenceObject function (IPB_Session) 158
RegisterClass, Windows method 32
Release function (IPB_Session) 172
ReleaseArrayInfo function (IPB_Session) 172
ReleaseDateString function (IPB_Session) 173
ReleaseDateTimeString function (IPB_Session) 173
ReleaseDecimalString function (IPB_Session) 174
ReleaseResultSetAccessor function (IPB_Session) 174
ReleaseString function (IPB_Session) 175
ReleaseTimeString function (IPB_Session) 176
ReleaseValue function (IPB_Session) 176
RemoveGlobalRef function (IPB_Session) 177
RemoveLocalRef function (IPB_Session) 178
RemoveProp function (IPB_Session) 178
result sets, accessing 72
RunApplication function (IPB_VM) 213

S
samples, on the Web 3
SDK, contents 8
SetArray function (IPB_Value) 208
SetArrayField function (IPB_Session) 182
SetArrayGlobalVar function (IPB_Session) 184
SetArrayItemToNull function (IPB_Session) 186
SetArrayItemValue function (IPB_Session) 186
SetArraySharedVar function (IPB_Session) 185
SetAt function (PBArrayAccessor template class) 230
SetAt function (PBBoundedArrayArrayCreator template class) 233
SetAt function (PBBoundedObjectArrayCreator template class) 235
SetAt function (PBOBJECTArrayArrayCreator template class) 237
SetAt function (PBUnboundedArrayArrayCreator template class) 239
SetAt function (PBUnboundedObjectArrayCreator template class) 240
SetBlob function (IPB_Session) 178
SetBlob function (IPB_Value) 208
SetBlobArrayItem function (IPB_Session) 180
SetBlobField function (IPB_Session) 182
SetBlobGlobalVar function (IPB_Session) 184
SetBlobSharedVar function (IPB_Session) 185
SetBool function (IPB_Value) 208
SetBlobArrayItem function (IPB_Session) 180
SetBoolField function (IPB_Session) 182
SetBoolGlobalVar function (IPB_Session) 184
SetBoolSharedVar function (IPB_Session) 185
SetByte function (IPB_Value) 208
SetByteArrayItem function (IPB_Session) 180
SetByteField function (IPB_Session) 182
SetByteGlobalVar function (IPB_Session) 184
SetByteSharedVar function (IPB_Session) 185
SetChar function (IPB_Value) 208
SetCharArrayItem function (IPB_Session) 180
SetCharField function (IPB_Session) 182
SetCharGlobalVar function (IPB_Session) 184
SetCharSharedVar function (IPB_Session) 185
SetData function (IPB_RSItemData interface) 95
SetDate function (IPB_Session) 187
SetDate function (IPB_Value) 208
SetDateArrayItem function (IPB_Session) 180
SetDateField function (IPB_Session) 182
SetDateGlobalVar function (IPB_Session) 184
SetDateSharedVar function (IPB_Session) 185
SetDateTime function (IPB_Session) 188
SetDateTime function (IPB_Value) 208
SetDateTimeArrayItem function (IPB_Session) 180
SetDateTimeField function (IPB_Session) 182
SetDateTimeGlobalVar function (IPB_Session) 184
SetDateTimeSharedVar function (IPB_Session) 185
SetDec function (IPB_Value) 208
SetDecArrayItem function (IPB_Session) 180
SetDecField function (IPB_Session) 182
SetDecimal function (IPB_Session) 189
SetDecSharedVar function (IPB_Session) 185
SetDouble function (IPB_Value) 208
SetDoubleArrayItem function (IPB_Session) 180
SetDoubleField function (IPB_Session) 182
SetDoubleGlobalVar function (IPB_Session) 184
SetDoubleSharedVar function (IPB_Session) 185
SetFieldToNull function (IPB_Session) 190
SetGlobalVarToNull function (IPB_Session) 190
SetInt function (IPB_Value) 208
SetIntArrayItem function (IPB_Session) 180
SetIntField function (IPB_Session) 182
SetIntGlobalVar function (IPB_Session) 184
SetIntSharedVar function (IPB_Session) 185
SetLong function (IPB_Value) 208
SetLongArrayItem function (IPB_Session) 180
SetLongField function (IPB_Session) 182
SetLongGlobalVar function (IPB_Session) 184
SetLongLong function (IPB_Value) 208
SetLongLongArrayItem function (IPB_Session) 180
SetLongLongField function (IPB_Session) 182
SetLongLongGlobalVar function (IPB_Session) 184
SetLongLongSharedVar function (IPB_Session) 185
SetNull function (IPB_RSItemData interface) 96
SetNullValue function (IPB_Value) 210
SetObject function (IPB_Value) 208
SetObjectArrayItem function (IPB_Session) 180
SetObjectField function (IPB_Session) 182
SetObjectGlobalVar function (IPB_Session) 184
SetObjectSharedVar function (IPB_Session) 185
SetPBString function (IPB_Value) 208
SetPBStringArrayItem function (IPB_Session) 180
SetPBStringField function (IPB_Session) 182
SetPBStringGlobalVar function (IPB_Session) 184
SetPBStringSharedVar function (IPB_Session) 185
SetProp function (IPB_Session) 192
SetReal function (IPB_Value) 208
SetRealArrayItem function (IPB_Session) 180
SetRealField function (IPB_Session) 182
SetRealGlobalVar function (IPB_Session) 184
SetRealSharedVar function (IPB_Session) 185
SetString function (IPB_Session) 194
SetString function (IPB_Value) 208
SetStringArrayItem function (IPB_Session) 180
SetStringField function (IPB_Session) 182
SetStringGlobalVar function (IPB_Session) 184
SetStringSharedVar function (IPB_Session) 185
SetTime function (IPB_Session) 196
SetTime function (IPB_Value) 208
SetTimeArrayItem function (IPB_Session) 180
SetTimeField function (IPB_Session) 182
SetTimeGlobalVar function (IPB_Session) 184
SetTimeSharedVar function (IPB_Session) 185
SetToNull function (PBArrayAccessor template class) 231
SetUint function (IPB_Value) 208
SetUintArrayItem function (IPB_Session) 180
SetUintField function (IPB_Session) 182
SetUintGlobalVar function (IPB_Session) 184
SetUintSharedVar function (IPB_Session) 185
SetUlong function (IPB_Value) 208
SetUlongArrayItem function (IPB_Session) 180
SetUlongField function (IPB_Session) 182
SetUlongGlobalVar function (IPB_Session) 184
SetUlongSharedVar function (IPB_Session) 185
SetValue function (IPB_Session) 197
Software Development Kit, contents 8
SplitDate function (IPB_Session) 197
SplitDateTime function (IPB_Session) 198
SplitTime function (IPB_Session) 198
system functions, calling 39
Index

T
ThrowException function (IPB_Session) 199
TriggerEvent function (IPB_Session) 200
triggering events 37
typographical conventions xi

U
UpdateField function (IPB_Session) 201

V
variables, using throughout a session 63
Visual C++ wizard 263
visual classes
creating instances 31
declaring 29
defining 29
describing 19
exporting methods 29
implementing 29
message processing 34
registering 32
visual extensions
about 5
building 28
creating instances 31
event processing 34
planning 28
using 30
Visual Studio wizards 263

W
WindowProc, in visual extensions 34
Windows messages
capturing 35
processing 38
wizard, for Visual C++ 263
wizards, for Visual Studio 263