

SYBASE®

PowerBuilder Native Interface Programmers
Guide and Reference

PowerBuilder®

11.5

DOCUMENT ID: DC37794-01-1150-01

LAST REVISED: September 2008

Copyright © 2008 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the [Sybase trademarks page](http://www.sybase.com/detail?id=1011207) at <http://www.sybase.com/detail?id=1011207>. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

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	25
	Adding an extension to a PowerBuilder target	25
	Using the extension	26
	Creating and using a visual extension	27
	Step 1: Decide on a feature to implement	27
	Step 2: Define the classes and functions in the extension	27
	Step 3: Declare visual classes and global functions	28
	Step 4: Implement native classes	28
	Step 5: Export methods to create class instances	28
	Step 6: Build and use a PBX	28
	Step 7: Use the visual extension in an application	29

	Creating visual class instances	29
	Event processing in visual extensions	32
	Using an event name with return type and arguments.....	33
	Using an event name with a PowerBuilder event ID	35
	Processing events sent to the parent of the window	36
	Calling PowerScript from an extension	37
	Example: Calling PowerBuilder functions.....	40
	Exception handling and debugging	41
CHAPTER 3	Creating Marshaler Extensions	43
	About marshaler extensions.....	43
	Developing the PowerBuilder extension	44
	Step 1: Describe the extension	45
	Step 2: Implement the creator class.....	46
	Step 3: Implement the marshaler class	49
	Generating proxies for Java classes	51
	Calling the Java class from PowerBuilder	52
CHAPTER 4	Exchanging Data with PowerBuilder	53
	About exchanging data with PowerBuilder.....	53
	Passing values between extensions and the PBVM	53
	PBCallInfo structure	54
	IPB_Arguments interface	54
	IPB_Value interface.....	55
	Using the IPB_Session interface.....	57
	Saving data from IPB_Value to a local variable	59
	Using variables throughout a session	61
	Handling enumerated types	62
CHAPTER 5	Calling PowerBuilder from C++.....	63
	About calling PowerScript from C++ applications	63
	Calling PowerBuilder objects from C++	64
	Creating a PowerBuilder object to be called from C++	64
	Getting the signature of a function	65
	Creating the C++ application.....	66
	Running the C++ application	70
	Accessing result sets	70
	Processing PowerBuilder messages in C++	71
	Example	71
	More PBNI possibilities	76

PART 2

REFERENCE

CHAPTER 6	PBNI Types and Return Values	81
	PowerBuilder to PBNI datatype mappings	81
	Types for access to PowerBuilder data.....	82
	PBNI enumerated types	82
	Error return values	84
CHAPTER 7	PBNI Interfaces, Structures, and Methods	85
	Header file contents	86
	Class and interface summary.....	86
	IPB_Arguments interface	88
	GetAt	88
	GetCount	89
	IPB_ResultSetAccessor interface	90
	AddRef	90
	GetColumnCount.....	90
	GetColumnMetaData.....	91
	GetItemData	92
	GetRowCount.....	92
	Release	93
	IPB_RSItemData interface	93
	SetData	93
	SetNull.....	94
	IPB_Session interface.....	94
	AcquireArrayItemValue	100
	AcquireValue	101
	Add<type>Argument	102
	AddGlobalRef	104
	AddLocalRef	104
	ClearException	105
	CreateResultSet	105
	FindClass	109
	FindClassByClassID.....	109
	FindGroup	110
	FindMatchingFunction	110
	FreeCallInfo.....	112
	Get<type>ArrayItem	112
	Get<type>Field.....	113
	Get<type>GlobalVar.....	115
	Get<type>SharedVar	116
	GetArrayInfo	117
	GetArrayItemType	118

GetArrayLength	119
GetBlob	120
GetBlobLength	120
GetClass.....	121
GetClassName	122
GetCurrGroup.....	122
GetDateString.....	123
GetDateTimeString.....	123
GetDecimalString	124
GetEnumItemName.....	124
GetEnumItemValue	125
GetException	126
GetFieldID	126
GetFieldName	127
GetFieldType	128
GetGlobalVarID	128
GetGlobalVarType.....	129
GetMarshaler.....	130
GetMethodID	131
GetMethodIDByEventID	132
GetNativeInterface	133
GetNumOfFields.....	134
GetPBAAnyArrayItem	134
GetPBAAnyField.....	135
GetPBAAnyGlobalVar.....	137
GetPBAAnySharedVar	137
GetProp	138
GetResultSetAccessor	138
GetSharedVarID.....	139
GetSharedVarType	140
GetString	141
GetStringLength	142
GetSuperClass	142
GetSystemClass.....	143
GetSystemGroup.....	143
GetTimeString	144
HasExceptionThrown	144
InitCallInfo	145
InvokeClassFunction	146
InvokeObjectFunction.....	147
IsArrayItemNull.....	148
IsAutoInstantiate.....	148
IsFieldArray	148
IsFieldNull.....	149

IsFieldObject	150
IsGlobalVarArray	150
IsGlobalVarNull	151
IsGlobalVarObject	152
IsNativeObject	153
IsSharedVarArray	154
IsSharedVarNull	154
IsSharedVarObject	155
NewBlob	155
NewBoundedObjectArray	156
NewBoundedSimpleArray	157
NewDate	158
NewDateTime	158
NewDecimal	159
NewObject	160
NewProxyObject	160
NewString	161
NewTime	162
NewUnboundedObjectArray	162
NewUnboundedSimpleArray	163
PopLocalFrame	164
ProcessPBMessage	164
PushLocalFrame	166
Release	166
ReleaseArrayInfo	166
ReleaseDateString	167
ReleaseDateTimeString	167
ReleaseDecimalString	168
ReleaseResultSetAccessor	168
ReleaseString	168
ReleaseTimeString	169
ReleaseValue	170
RemoveGlobalRef	171
RemoveLocalRef	171
RemoveProp	172
Set<type>ArrayItem	172
Set<type>Field	174
Set<type>GlobalVar	175
Set<type>SharedVar	176
SetArrayItemToNull	178
SetArrayItemValue	178
SetBlob	179
SetDate	179
SetDateTime	180

SetDecimal	180
SetFieldToNull	181
SetGlobalVarToNull	182
SetMarshaler	182
SetProp	183
SetSharedVarToNull	185
SetString	186
SetTime	187
SetValue	188
SplitDate	189
SplitDateTime	189
SplitTime	190
ThrowException	190
TriggerEvent	191
UpdateField	192
IPB_Value interface	193
Get<type>	194
GetClass	195
GetType	195
IsArray	196
IsByRef	196
IsEnum	197
IsNull	197
IsObject	198
Set<type>	198
SetToNull	200
IPB_VM interface	201
CreateSession	201
RunApplication	202
IPBX_Marshaler interface	204
Destroy	204
GetModuleHandle	205
InvokeRemoteMethod	206
IPBX_NonVisualObject interface	207
IPBX_UserObject interface	208
Destroy	208
Invoke	208
IPBX_VisualObject interface	210
CreateControl	210
GetEventID	212
GetWindowClassName	214
PBArrayInfo structure	215
PBCallInfo structure	215

PB_DateData structure	216
PB_DateTimeData structure	216
PB_TimeData structure	216
PBX_DrawItemStruct structure	217
PBArrayAccessor template class	218
GetAt	218
IsNull	218
SetAt.....	219
SetToNull.....	220
PBBoundedArrayCreator template class	221
GetArray	221
SetAt.....	222
PBBoundedObjectArrayCreator class	224
GetArray	224
SetAt.....	224
PBObjectArrayAccessor class	225
GetAt	225
SetAt.....	226
PBUnboundedArrayCreator template class	227
GetArray	227
SetAt.....	227
PBUnboundedObjectArrayCreator class.....	228
GetArray	228
SetAt.....	229
Exported methods	230
PBX_CreateNonVisualObject.....	230
PBX_CreateVisualObject	232
PBX_DrawVisualObject.....	233
PBX_GetDescription	234
PBX_InvokeGlobalFunction	237
PBX_Notify	239
Method exported by PowerBuilder VM.....	240
PB_GetVM	240

CHAPTER 8	PBNI Tool Reference	243
	pbsig115.....	243
	pbx2pbd115	246

PART 3

APPENDIX

APPENDIX A	Using the Visual Studio Wizards.....	251
	Where the wizards are installed	251
	Generating a PBNI project	253
	Setting project options.....	253
	Building and using the PBX.....	254
Index		255

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, the SyBooks CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

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

- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

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

Conventions

The formatting conventions used in this manual are:

Formatting example	To indicate
Retrieve and Update	When used in descriptive text, this font indicates: <ul style="list-style-type: none">• Command, function, and method names• Keywords such as true, false, and null• Datatypes such as integer and char• Database column names such as emp_id and f_name• User-defined objects such as dw_emp or w_main
<i>variable or file name</i>	When used in descriptive text and syntax descriptions, oblique font indicates: <ul style="list-style-type: none">• Variables, such as <i>myCounter</i>• Parts of input text that must be substituted, such as <i>pblname.pbd</i>• File and path names
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.”
<code>dw_1.Update()</code>	Monospace font indicates: <ul style="list-style-type: none">• Information that you enter in a dialog box or on a command line• Sample script fragments• Sample output fragments

If you need help

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

PART 1

Programmers Guide

This part provides an introduction to the PowerBuilder Native Interface and a guide to creating PowerBuilder extensions and interacting with PowerBuilder.

About this chapter

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

Contents

Topic	Page
About PBNI	3
The elements of PBNI	6
The PBNI SDK	8
Comparing PBNI and JNI	10

About PBNI

PBNI is a standard programming interface that enables developers to extend the functionality of PowerBuilder. Using PBNI, 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 PBNI, Java applications can also communicate with the PBVM.

Use with .NET targets

You can use the built-in Web services client extension (*pbwsclient115.pbx*) in applications that you plan to deploy to .NET as a PowerBuilder .NET Windows Forms application. You *cannot* use any other PBNI extensions in a .NET target.

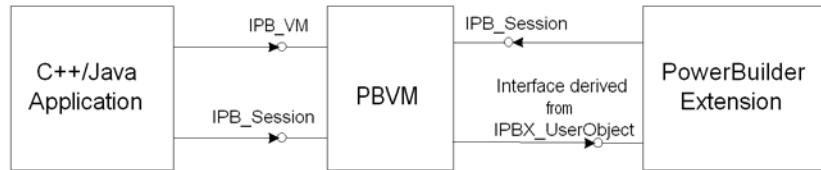
Code samples

This documentation contains two complete but very simple examples that illustrate some basic principles of using the PowerBuilder Native Interface (PBNI): “Nonvisual extension example” on page 11 and “Creating a PowerBuilder object to be called from C++” on page 64. For more real-world examples, see the PBNI section of the PowerBuilder CodeXchange Web site at <http://powerbuilder.codeXchange.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

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 (winproc) 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 PBNI 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 PBNI

To enable the features described in the previous section, PBNI 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, "PBNI 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.

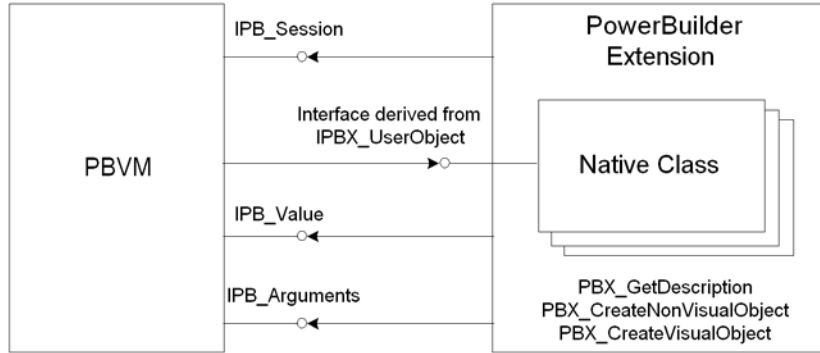
Helper classes

Several helper classes, such as PObjectCreator, PBArrayAccessor, and PEventTrigger, make it easier to program with PBNI.

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 PBNI SDK

When you install PowerBuilder, the Software Development Kit (SDK) for PBNI is installed in the *PowerBuilder 11.5\SDK\PBNI* directory. The SDK tools, *pbsig115* and *pbx2pbd115*, 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 PBNI SDK

Component	Description
<i>pbx2pbd115.exe</i>	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 PBNI functions. The DLL is usually called a PBX and can be given the suffix <i>.pbx</i> .
<i>pbsig115.exe</i>	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.
<i>include\pbni.h</i>	A header file that defines the structures and interfaces used to build PowerBuilder extensions.
<i>include\pbarrray.h</i>	A header file that contains helper classes that make it easier to create arrays and access data in them.

Component	Description
<i>include\pbfield.h</i>	A header file that contains helper classes that make it easier to access data in fields.
<i>include\pbtraits.h</i>	A header file used by <i>pbarray.h</i> and <i>pbfield.h</i> that provides specializations for the <i>pbvalue_type</i> enumerated types.
<i>include\pbext.h</i>	A header file that defines the functions that PowerBuilder extension functions must export.
<i>include\pbevtid.h</i>	A header file that maps the PowerBuilder event IDs to event names for use in visual extensions.
<i>include\pbnimd.h</i>	A header file that defines machine-dependent datatypes used in <i>pbni.h</i> .
<i>include\pbrsa.h</i>	A header file that defines interfaces and structures used to access DataWindow and DataStore data.
<i>src\pbarray.cpp</i>	A source file that must be added to your project if you want to use the following helper classes defined in <i>pbarray.h</i> : PBArrayAccessor PBOBJECTArrayAccessor PBBoundedArrayCreator PBBoundedObjectArrayCreator PBUnboundedArrayCreator PBUnboundedObjectArrayCreator
<i>src\pbfuninv.cpp</i>	A source file that must be added to your project if you want to use the following helper classes defined in <i>pbni.h</i> : PBGlobalFunctionInvoker PBOBJECTFunctionInvoker PBEventTrigger
<i>src\pbobject.cpp</i>	A source file that must be added to your project if you want to use the following helper class defined in <i>pbni.h</i> : PBOBJECTCreator.
<i>wizards\VCProjects 8.0</i>	A Microsoft Visual Studio 2005 wizard that makes it easier for you to create PBNI projects.
<i>wizards\VCProjects 7.1</i>	A Microsoft Visual Studio .NET 2003 wizard that makes it easier for you to create PBNI projects.
<i>wizards\VCProjects 7.0</i>	A Microsoft Visual Studio .NET 2002 wizard that makes it easier for you to create PBNI projects.
<i>wizards\VCWizards</i>	Files required by the Visual Studio wizards.
<i>pbni115.hlp</i> , <i>pbni115.cnt</i>	Help files for PBNI.

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 `pbsig115` 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.

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

Topic	Page
Nonvisual extension example	11
Creating a PowerBuilder extension	17
Adding an extension to a PowerBuilder target	25
Using the extension	26
Creating and using a visual extension	27
Creating visual class instances	29
Event processing in visual extensions	32
Calling PowerScript from an extension	37
Exception handling and debugging	41

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.”

For more realistic examples, see the PowerBuilder CodeXchange Web site at <http://powerbuilder.codexchange.sybase.com>.

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*:

```
#include "pbext.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);

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



```
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 PBNI method.

The following is the code for *main.cpp*:

```
#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
PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[]={
        "class pbadd from nonvisualobject \n" \
        "function int f_add(int a,int b)\n" \
        "end class \n"
    };
    return desc;
}
```

```

// 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()
{
}

```

```
// implement the class's f_add method
int pbadd:: f_add(IPB_Session* session, puint arg1,
                puint 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 11.5\SDK\PBNI* is in your include path. For this example, the generated DLL is called *pbadd.pbx*.

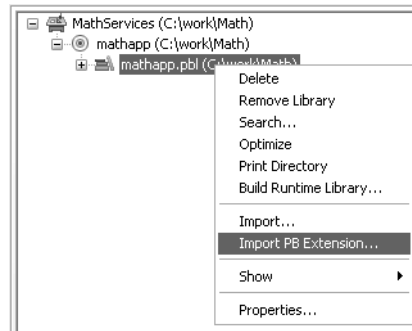
Using the extension in PowerBuilder

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.

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



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

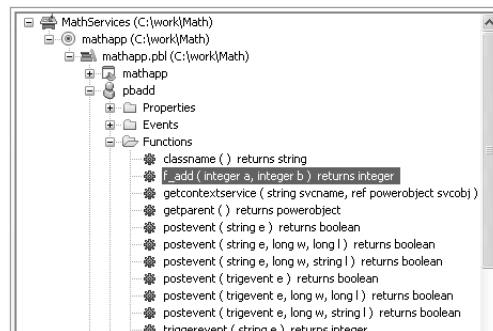
❖ **To invoke the `f_add` function in PowerBuilder:**

- Create a new window called `w_add`, and add three single-line edit boxes and a command button to it.
- 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 27. 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 27.

If the extension declares global functions, it must also export the `PBX_InvokeGlobalFunction` method.

For every native class, you must implement two PBNI methods, `Invoke` and `Destroy`, in addition to the methods the class will provide. The use of these PBNI 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.

For examples of PowerBuilder extensions, see the PowerBuilder CodeXchange Web site at <http://powerbuilder.codexchange.sybase.com>.

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 234.

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:

```
PBEXPORT 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 27.

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:

```
#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,
        pobject obj,
        pbmethodID mid,
        PBCallInfo *ci
    );
    void Destroy();

private:
    void funcA(IPB_Session* session, pobject obj,
        PBCallInfo* ci);
    void funcB(IPB_Session* session, pobject 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`:

```
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:

```
PBXRESULT MyClass::Invoke(IPB_Session *session,
pbject 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;
}

```

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:

```

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 (strcmp(functionName, "bitand") == 0)
    {
        ci->returnValue->SetInt(arg1 & arg2);
    }else if (strcmp(functionName, "bitor") == 0)
    {
        ci->returnValue->SetInt(arg1 | arg2);
    }else if (strcmp(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 29.
- 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:

```
PBXEXPORT PBXRESULT PBXCALL PBX_CreateNonVisualObject (
    IPB_Session * session,
    pobject 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, "classtree" ) == 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 11.5\SDK\PBNI\include*, 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.

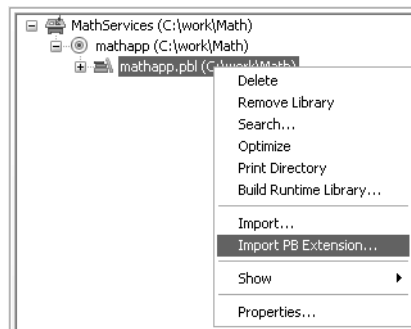
Now you are ready to use the extension in a PowerBuilder application.

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 *pbx2pbd115* on page 246.

❖ 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:

```
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:

```
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 71.

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:

```
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 32.

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 29.

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 25 and “Adding an extension to a PowerBuilder target” on page 25 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 29. 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:

```
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;
};
```

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:

```
void CVisualExt::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:

```
void CVisualExt::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:

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

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

        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
            break;

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

Implementing CreateControl

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
                           // 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;
}
```

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:

```
event returnType eventName(args_desc) newline
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:

```
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 setttext(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:

```
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);
EndPoint(hwnd, &ps);
}
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:

```
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:

```
PBEXPORT 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 settex(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_LBUTTONDOWNBLCLK` 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.

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.

```
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 the `GetMethodID` or `FindMatchingFunction` method.

Using `GetMethodID`

To get the function's ID using the `GetMethodID` method, you need the function's signature:

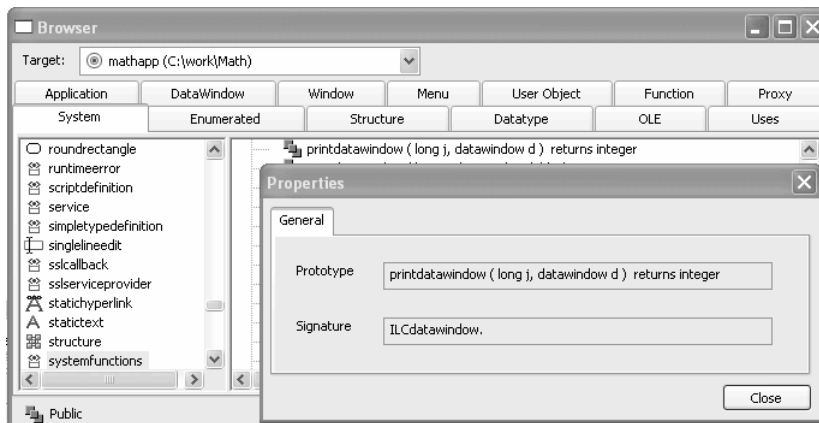
```

PbmethodID GetMethodID(pbclass cls, LPCTSTR
    methodName, PBRoutineType rt, LPCTSTR signature);

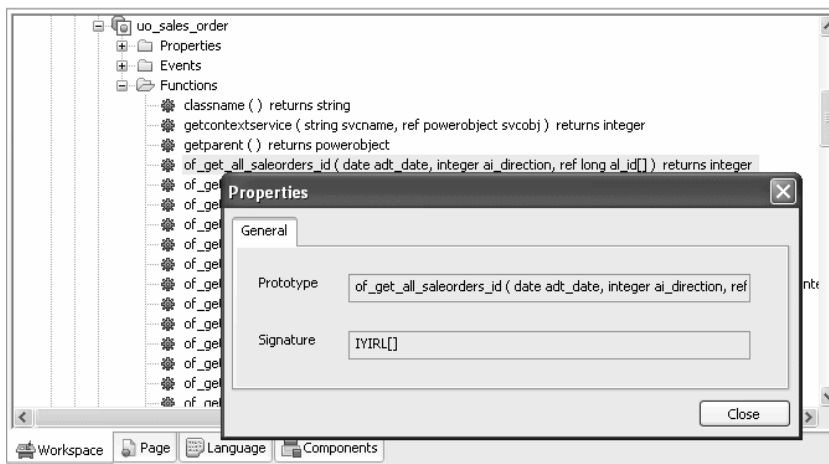
```

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:



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:



Consider this function:

```
of_get_trans ( ref transaction atr_trans ) returns
(none)
```

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

You can use the `pbsig115` command-line tool to obtain a function's signature. However, the `pbsig115` 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 `pbsig115`, and an explanation of all the formats used in the signature, see `pbsig115` on page 243.

Using FindMatchingFunction

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`:

```
FindMatchingFunction(pbclass cls, 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:

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

Invoking PowerBuilder functions

The following methods are those you use most frequently to invoke PowerBuilder functions. For descriptions of each method, see `IPB_Session` interface on page 94.

```
PbmethodID GetMethodID(pbclass cls, LPCTSTR methodName,
                      PBRoutineType rt, LPCTSTR signature, pbboolean publicly)
PBXRESULT InitCallInfo(pbclass cls, pbmethodID mid, PBCallInfo *ci)
void FreeCallInfo(PBCallInfo *ci)
PBXRESULT Add<Type>Argument(PBCallInfo *ci, PBType v);
PBXRESULT InvokeClassFunction(pbclass cls, pbmethodID mid,
                              PBCallInfo *ci)
```

```
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.

```
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:

```

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.

About this chapter

This chapter describes how to create marshaler extensions.

Contents

Topic	Page
About marshaler extensions	43
Developing the PowerBuilder extension	44
Generating proxies for Java classes	51
Calling the Java class from PowerBuilder	52

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:

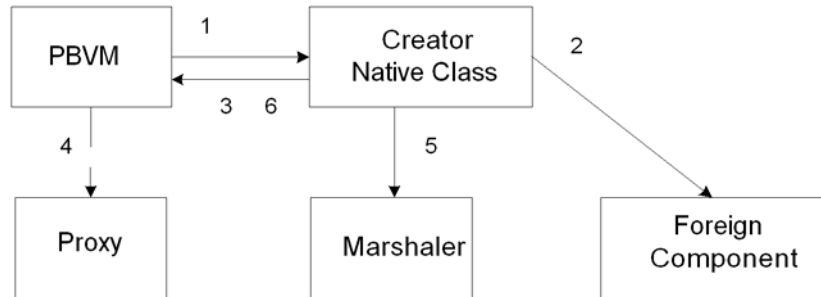
- Developing the PowerBuilder extension
- Generating proxies for Java classes
- Calling the Java class from PowerBuilder

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

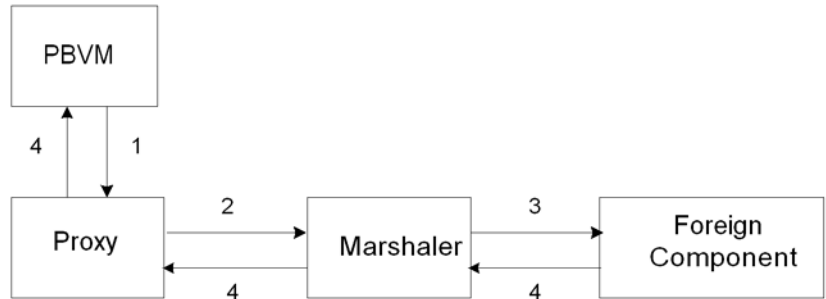
A PowerBuilder marshaler extension usually provides a native class that acts as a creator. This class defines a function that creates an instance of the foreign component that is specified in the parameters passed into the function (1). If it succeeds in creating an instance of the foreign component (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 send 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`:

```

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;
}
  
```

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:

```
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;
        }
    }
}
```

```
        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, jobj);

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

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

ci->returnValue->SetLong(kSuccessful);

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:

```
#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*:

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

extern pbulong g_dll_hModule;

pbulong JavaMarshaler::GetModuleHandle()
{
    return g_dll_hModule;
}
```

```

//*****
//  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
            }
        }
    }
}

```

```

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

return PBX_OK;
}

void JavaMarshaler::Destroy()
{
    delete this;
}

```

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:

```

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"

```

```
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 though JNI and returns the result to PowerBuilder.

Exchanging Data with PowerBuilder

About this chapter

This chapter describes how PBNI extensions exchange data with PowerBuilder.

Contents

Topic	Page
About exchanging data with PowerBuilder	53
Passing values between extensions and the PBVM	53
Using the IPB_Session interface	57
Saving data from IPB_Value to a local variable	59
Using variables throughout a session	61
Handling enumerated types	62

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:

```
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:

```
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:

```
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):

```
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:

```
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 54, 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:

```
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>(<pdtype> 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`:

```
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 <pdtype> 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`:

```
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 `parray`), 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 94.

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 PBN1 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.

```
// boolean isNull[], pbject myobj,
// and pdate* d_date arguments passed in
pclass cls;
pmethodID 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.

This applies to the following pointer value datatypes, as well as to the pbaray 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.

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 PowerShell, 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.

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

Topic	Page
About calling PowerScript from C++ applications	63
Calling PowerBuilder objects from C++	64
Accessing result sets	70
Processing PowerBuilder messages in C++	71
More PBNI possibilities	76

About calling PowerScript from C++ applications

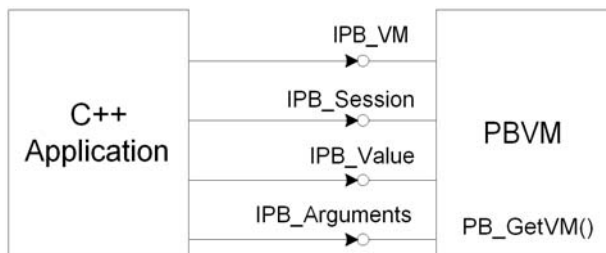
If you have a PowerBuilder custom class user object that performs intensive programming that would be useful to an application that you need to write in C++, you can access the object directly from the C++ application using PBNI. You do not need to make the user object into a COM component or automation 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 *pbvm115.dll* with the Windows LoadLibrary function, get a pointer to the IPB_VM interface by calling the PB_GetVM function exported by *pbvm115.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.

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.

- In the Function prototype window, create a function with this signature:

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

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

Getting the signature of a function

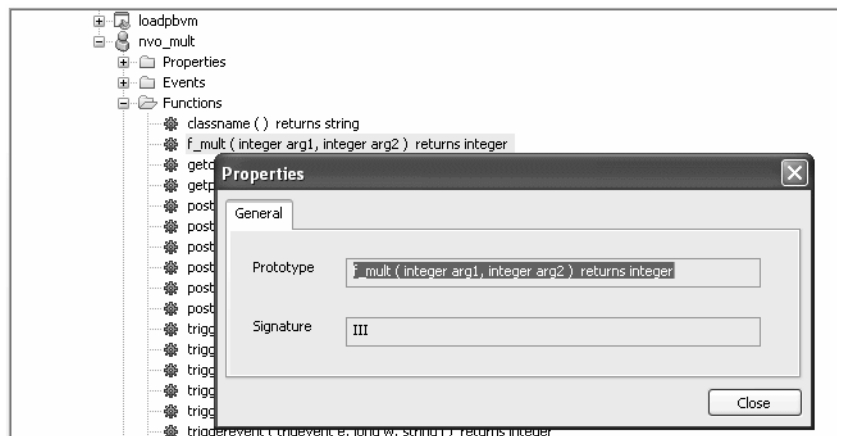
To write the C++ code that invokes the `f_mult` function, you need to obtain its method ID. The method ID is used to initialize the `PBMethodInfo` structure and to invoke the function. There are two `IPB_Session` functions that return a method ID: `GetMethodID`, which takes a signature, and `FindMatchingFunction`, 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 37.

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 `pbsig115` tool.

Getting a signature
from the System Tree

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:



Getting a signature using pbsig115

To get the signature of `f_mult` with `pbsig115`, type the following at a command prompt:

```
pbsig115 d:\pb1s\loadpbvm.pbl
```

In the output of `pbsig115`, 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 `pbsig115` tool and the format of method signatures, see `pbsig115` on page 243.

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 11.5\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.

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

```
#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("pbvm115.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");
```

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:

```
// 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 65 to the GetMethodID function:

```
// 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:

```
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.

```
// 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:

```
// 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
puint ret = ci.returnValue->GetInt();
fprintf(stderr, "The product of 123 and 45 is %i\n",
    ret);
```

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:

```
// 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 105 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.

ProcessPBMessage
must be called
repeatedly

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:

```

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

```

Example

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.

Calling
ProcessPBMMessage

The call to `ProcessPBMMessage` is in a loop in the `WinMain` function:

```
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 ProcessPBMMessage
            if (session)
                session->ProcessPBMMessage();
        }
    }
    catch(...)
    {
        MessageBox(NULL, "Exception occurs",
            "Exception", MB_OK);
    }
    return msg.wParam;
}
```

Loading the PBVM
and triggering an
event

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:

```
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("pbvm115.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 ProcessPBMMessage

You can test the `ProcessPBMMessage` 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 `ProcessPBMMessage` 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->ProcessPBMMessage();
```

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 PBNI possibilities

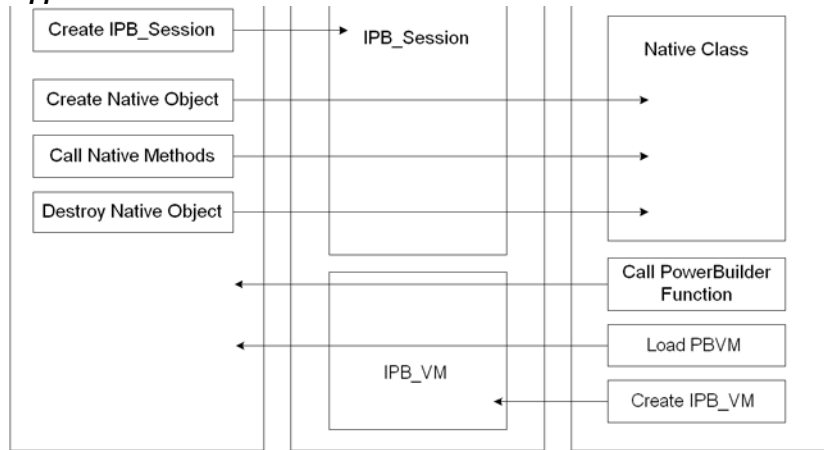
The ability to create visual, nonvisual, and marshaler extensions, and to call PowerBuilder objects from external C++ applications, opens up numerous opportunities to combine these capabilities to develop more complex applications.

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 PBNI, 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 PBNI. 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 PBNI.

There is a sample that illustrates these techniques on the PowerBuilder CodeXchange Web site at <http://powerbuilder.codexchange.sybase.com>.

Reference

This part contains reference information for PBNI datatypes, interfaces, and tools.

About this chapter

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

Contents

Topic	Page
PowerBuilder to PBNI datatype mappings	81
Types for access to PowerBuilder data	82
PBNI enumerated types	82
Error return values	84

PowerBuilder to PBNI datatype mappings

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

Table 6-1: Predefined datatype mappings

PowerBuilder datatype	Predefined datatype
Int	pbint
UInt	pbuint
Byte	pbbyte
Long	pblong
Longlong	pblonglong
Ulong	pbulong
Boolean	pbboolean
Real	pbreal
Double	pbdouble
Decimal	pbdec
Date	pbdate
Time	pbtime
DateTime	pbdatetime
Char	pbchar
Blob	pbblob

PowerBuilder datatype	Predefined datatype
String	pbstring
Powerobject	pbobject

Types for access to PowerBuilder data

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

Table 6-2: Types for access to PowerBuilder data

Datatype	Description
pbgroup	Used to access PowerBuilder group information. A group is a container of PowerBuilder classes.
pbclass	Used to access PowerBuilder class definition information.
pbmethodID	Used to access the method ID of a PowerBuilder global or member function.
pbfieldID	Used to access an instance variable.
pbarray	Used to access array information and data items.

PBNI enumerated types

Enumerated types for PowerBuilder groups

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

Table 6-3: Enumerated types for PowerBuilder groups

Value	PowerBuilder object
<code>pbgroup_application</code>	Application
<code>pbgroup_datawindow</code>	DataWindow definition
<code>pbgroup_function</code>	Global function
<code>pbgroup_menu</code>	Menu
<code>pbgroup_proxy</code>	Proxy definition for a remote object
<code>pbgroup_structure</code>	PowerBuilder structure type
<code>pbgroup_userobject</code>	PowerBuilder user object
<code>pbgroup_window</code>	Window
<code>pbgroup_unknown</code>	Unknown group

Enumerated types for PowerBuilder values

The *pbvalue_type* enumerated types are used in methods such as the *IPB_Value GetType* method and the *IPB_Session NewUnboundedSimpleArray* method to identify the type of PowerBuilder data.

Table 6-4: Enumerated types for PowerBuilder values

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

Enumerated types for PowerBuilder routines

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

Value	Routine type
PBRT_FUNCTION	Function
PBRT_EVENT	Event

Error return values

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

Table 6-6: PBXResult return values

Value of PBXResult	Error code
PBX_OK	0
PBX_SUCCESS	0
PBX_FAIL	-1
PBX_E_NO_REGISTER_FUNCTION	-1
PBX_E_REGISTRATION_FAILED	-2
PBX_E_BUILD_GROUP_FAILED	-3
PBX_E_INVALID_ARGUMENT	-4
PBX_E_INVOKE_METHOD_INACCESSABLE	-5
PBX_E_INVOKE_WRONG_NUM_ARGS	-6
PBX_E_INVOKE_REFARG_ERROR	-7
PBX_E_INVOKE_METHOD_AMBIGUOUS	-8
PBX_E_INVOKE_FAILURE	-9
PBX_E_MISMATCHED_DATA_TYPE	-10
PBX_E_OUTOF_MEMORY	-11
PBX_E_GET_PBVM_FAILED	-12
PBX_E_NO_SUCH_CLASS	-13
PBX_E_CAN_NOT_LOCATE_APPLICATION	-14
PBX_E_INVALID_METHOD_ID	-15
PBX_E_READONLY_ARGS	-16
PBX_E_ARRAY_INDEX_OUTOF_BOUNDS	-100

PBNI Interfaces, Structures, and Methods

About this chapter

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

Contents

Topic	Page
Header file contents	86
Class and interface summary	86
IPB_Arguments interface	88
IPB_ResultSetAccessor interface	90
IPB_RSItemData interface	93
IPB_Session interface	94
IPB_Value interface	193
IPB_VM interface	201
IPBX_Marshaler interface	204
IPBX_NonVisualObject interface	207
IPBX_UserObject interface	208
IPBX_VisualObject interface	210
PBArrayInfo structure	215
PBCallInfo structure	215
PB_DateData structure	216
PB_DateTimeData structure	216
PB_TimeData structure	216
PBX_DrawItemStruct structure	217
PBArrayAccessor template class	218
PBBoundedArrayCreator template class	221
PBBoundedObjectArrayCreator class	224
PBObjectArrayAccessor class	225
PBUnboundedArrayCreator template class	227
PBUnboundedObjectArrayCreator class	228
Exported methods	230
Method exported by PowerBuilder VM	240

Header file contents

	PBNI classes and interfaces are defined in a set of header files.
<code>pbni.h</code>	The classes, structures, and methods defined in the header file <code>pbni.h</code> allow PowerBuilder extension modules to interact with PowerBuilder. This file also includes the <code>pbarray.h</code> , <code>pbfield.h</code> , and <code>pbnimd.h</code> header files.
<code>pbarray.h</code> , <code>pbfield.h</code> , <code>pbtraits.h</code> , and <code>pbnimd.h</code>	<code>pbarray.h</code> contains helper classes that make it easier to create arrays and access data in them. <code>pbfield.h</code> contains a helper class that makes it easier to access fields. Both header files rely on <code>pbtraits.h</code> , which provides specializations for the Value enumerated types. <code>pbnimd.h</code> contains machine-specific datatype definitions. These files should not be included directly in your code.
<code>pbext.h</code>	The classes, structures, and methods defined in the header file <code>pbext.h</code> must be implemented in PowerBuilder extension modules to allow PowerBuilder applications to use the extension modules. <code>pbext.h</code> includes <code>pbni.h</code> and <code>pbevtid.h</code> .
<code>pbevtid.h</code>	<code>pbevtid.h</code> 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 32.
<code>pbrsa.h</code>	<code>pbrsa.h</code> 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 PBNI. 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:

- `PBArrayInfoHolder` and `PBCallInfoHolder` – used to hold a `PBArrayInfo` or `PBCallInfo` variable and release it when it is out of scope
- `PBEventTrigger`, `PBObjectFunctionInvoker`, and `PBGlobalFunctionInvoker` – used to trigger events and call object and global functions

Table 7-1: PBNI class and interface summary

Object	Description	Defined in
IPB_Arguments interface	Used to access the arguments of the PBCallInfo structure.	<i>pbni.h</i>
IPB_ResultSetAccessor interface	Used to access data in a DataWindow or DataStore.	<i>pbrsa.h</i>
IPB_RSItemData interface	Used to set data values in a result set from a DataWindow or DataStore.	<i>pbrsa.h</i>
IPB_Session interface	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.	<i>pbni.h</i>
IPB_Value interface	Used to hold PowerBuilder data, IPB_Value contains information about each variable, including its type, null flag, access privileges, array or simple type, and reference type.	<i>pbni.h</i>
IPB_VM interface	Used to load PowerBuilder applications in third-party applications and interoperate with the PowerBuilder virtual machine (PBVM).	<i>pbni.h</i>
PBArrayInfo structure	Used to hold information about arrays.	<i>pbni.h</i>
PBCallInfo structure	Used to hold arguments and return type information in function calls between PBNI and PowerBuilder.	<i>pbni.h</i>
PB_DateData structure	Used to pass data of type Date in the SetData function in the IPB_RSItemData interface.	<i>pbrsa.h</i>
PB_DateTimeData structure	Used to pass data of type DateTime in the SetData function in the IPB_RSItemData interface.	<i>pbrsa.h</i>
PB_TimeData structure	Used to pass data of type Time in the SetData function in the IPB_RSItemData interface.	<i>pbrsa.h</i>
PBX_DrawItemStruct structure	Used to hold the properties of an external visual control that you want to draw using the PBX_DrawVisualObject function.	<i>pbext.h</i>
PBArrayAccessor template class	Used to access items in an array.	<i>pbarrray.h</i>
PBObjectArrayAccessor class	Used to access items in an object array.	<i>pbarrray.h</i>
PBBoundedArrayCreator template class	Used to create bounded arrays.	<i>pbarrray.h</i>
PBBoundedObjectArrayCreator class	Used to create bounded object arrays.	<i>pbarrray.h</i>
PBBoundedObjectArrayCreator class	Used to create unbounded arrays.	<i>pbarrray.h</i>
PBUnboundedObjectArrayCreator class	Used to create unbounded object arrays.	<i>pbarrray.h</i>

Object	Description	Defined in
IPBX_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.	<i>pbext.h</i>
IPBX_NonVisualObject interface	Inherits from IPBX_UserObject and is the direct ancestor class of nonvisual PowerBuilder native classes.	<i>pbext.h</i>
IPBX_UserObject interface	The ancestor class of PowerBuilder native classes. It has two functions, Destroy and Invoke.	<i>pbext.h</i>
IPBX_VisualObject interface	Inherits from IPBX_UserObject and is the direct ancestor class of visual PowerBuilder native classes.	<i>pbext.h</i>
Exported methods	Some exported methods <i>must</i> be implemented in PowerBuilder extension modules.	<i>pbext.h</i>
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.	<i>pbni.h</i>

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*)

Argument	Description
<i>index</i>	A valid index into the PBCallInfo structure

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:

```

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 pobject 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
<i>columnNum</i>	The number of the column for which you want to obtain metadata
<i>columnName</i>	The name of the specified column
<i>type</i>	A pointer to the type of the specified column
<i>width</i>	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:

```
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`

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*)

Argument	Description
<i>row</i>	The row number of the cell
<i>col</i>	The column number of the cell
<i>data</i>	A pointer to an IPB_RSItemData structure

Return value Boolean.

Examples This example stores the data in the first row and column in the IPB_RSItemData structure sd:

```
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*:

```
*numRows = d_rsAccessor->GetRowCount();
```

See also CreateResultSet
 GetColumnCount
 GetColumnMetaData
 GetItemData

Release

Description	When you call the <code>CreateResultSet</code> function of interface <code>IPB_Session</code> , you need to pass an argument of type <code>IPB_ResultSetAccessor</code> . The <code>AddRef</code> function is called on that argument and the <code>Release</code> function is called when the pobject is destroyed.
Syntax	<code>Release ()</code>
Return value	None.
See also	<code>AddRef</code> <code>CreateResultSet</code>

IPB_RSItemData interface

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

SetData

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

Argument	Description
<i>len</i>	The length of the data
<i>data</i>	A void pointer to the address of the data

Return value	None.
Usage	<p>If the cell datatype is:</p> <ul style="list-style-type: none"> • string and decimal, the address points to a string • date, the address points to a <code>PB_DateData</code> structure • time, the address points to a <code>PB_TimeData</code> structure • datetime, the address points to a <code>PB_DateTimeData</code> structure • another datatype, the address points to data of the corresponding type

See also GetItemData
 SetNull
 PB_DateData structure
 PB_DateTimeData structure
 PB_TimeData structure

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 7-2: IPB_Session methods by category

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

Purpose	Method	Description
	PushLocalFrame	Pushes a local reference frame onto the current native method stack frame.
	RemoveGlobalRef	Removes a global reference to the specified PowerBuilder object.
	RemoveLocalRef	Removes a local reference to the specified PowerBuilder object.
Managing shared properties	GetProp	Retrieves a pointer to the data value of a variable that has been registered as a shared property for the current IPB session.
	RemoveProp	Removes the specified variable from the list of properties of the current IPB session.
	SetProp	Adds a new variable to the list of properties of the current session or changes the value of an existing variable.
Handling the PowerBuilder message queue	ProcessPBMessage	Checks the PowerBuilder message queue and, if there is a message in the queue, attempts to process it.
Handling exceptions	ClearException	Clears the current PowerBuilder exception object.
	GetException	Obtains the current thrown exception object.
	HasExceptionThrown	Checks for the existence of an exception that has been thrown but not cleared.
	ThrowException	Throws a PowerBuilder exception or inherited exception, replacing the existing exception if one exists.
Passing arguments	Add<type>Argument	Adds an argument in a variable argument PowerBuilder call.
	FreeCallInfo	Frees memory allocated by InitCallInfo.
	InitCallInfo	Initializes the PBCallInfo structure.
Finding PowerBuilder classes and objects	FindGroup	Searches for a group with a given name and group type in the current library list.
	FindClass	Searches for a class with a given name within a given group.
	FindClassByClassID	Searches for a class with a given name and a given ID.
	GetClass	Returns the class handle of a PowerBuilder object.
	GetClassName	Returns the name of a class in lowercase.
	GetCurrGroup	Returns the name of the current group.
	GetSuperClass	Returns the base class of a class, if any.

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

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

Purpose	Method	Description
Working with binary large objects	GetBlob	Returns a pointer to the data buffer for a blob.
	GetBlobLength	Returns the length in bytes of blob data in a buffer.
	NewBlob	Creates a new blob and duplicates a buffer for the new blob data.
	SetBlob	Destroys the existing data in a blob and copies data into it from a buffer.
Working with decimal values	GetDecimalString	Converts decimal data in a pbdec object to a string.
	NewDecimal	Allocates resources for a new decimal data object.
	ReleaseDecimalString	Frees the memory acquired using GetDecimalString.
	SetDecimal	Converts a string to a decimal.
Working with date and time values	GetString	Converts data in a pbdate object to a string.
	GetDateTimeString	Converts data in a pbdatetime object to a string.
	GetTimeString	Converts data in a pbtime object to a string.
	NewDate	Creates a new pbdate data object.
	NewDateTime	Creates a new pbdatetime data object.
	NewTime	Creates a new pbtime data object.
	ReleaseDateString	Frees the memory acquired using GetString.
	ReleaseDateTimeString	Frees the memory acquired using GetDateTimeString.
	ReleaseTimeString	Frees the memory acquired using GetTimeString.
	SetDate	Resets the value of the specified pbdate object.
	SetDateTime	Resets the value of the specified pbdatetime object.
	SetTime	Resets the value of the specified pbtime object.
	SplitDate	Splits the specified pbdate object into a year, month, and day.
	SplitDateTime	Splits the specified pbdatetime object into a year, month, and day.
SplitTime	Splits the specified pbtime object into a year, month, and day.	

Purpose	Method	Description
Working with data values	AcquireArrayItemValue	Clones the data in the PBCallInfo structure in an array item and resets the IPB_Value pointer.
	AcquireValue	Clones the data in the PBCallInfo structure and resets the IPB_Value pointer.
	ReleaseValue	Frees the value acquired by the AcquireValue or AcquireArrayItemValue method.
	SetValue	Sets the value of one IPB_Value object to the value of another IPB_Value object
Working with fields	GetFieldID	Obtains the internal ID of a class instance variable.
	GetFieldName	Obtains the name of the specified field.
	GetFieldType	Obtains the datatype of a class instance variable.
	GetNumOfFields	Obtains the number of fields in the specified class.
	GetPBAnyField	Obtains the value of a variable of type Any.
	Get<type>Field	Obtains a pointer to the instance variable data for a specified variable.
	IsFieldArray	Returns true if the field contains an array, otherwise returns false.
	IsFieldNull	Returns true if the field contains a null value array, otherwise returns false.
	IsFieldObject	Returns true if the field contains a pobject, otherwise returns false.
	Set<type>Field	A set of datatype-specific functions. Sets the value of an instance field of an object.
	Set<type>Field	A set of datatype-specific functions. Sets the value of an instance field of an object.
	UpdateField	Refreshes a visual property of a PowerBuilder object.
Working with native classes	GetNativeInterface	Obtains a pointer to the interface of a native class.
	IsNativeObject	Determines whether a pobject is an instance of a native class.
Accessing result sets from DataWindows and DataStores	CreateResultSet	Creates a result set object using a pointer to an IPB_ResultSetAccessor object.
	GetResultSetAccessor	Obtains an interface through which you can read data from a result set.
	ReleaseResultSetAccessor	Releases the pointer obtained using GetResultSetAccessor.

Purpose	Method	Description
Working with marshaler extensions	GetMarshaler	Obtains the marshaler object associated with a proxy object.
	NewProxyObject	Creates a proxy for a remote object.
	SetMarshaler	Sets a marshaler that will be used to invoke remote methods and convert PowerBuilder data formats to the user's communication protocol.

AcquireArrayItemValue

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

Syntax AcquireArrayItemValue(pbararray *array*, pblong *dim*[])

Argument	Description
<i>array</i>	A valid pbararray structure.
<i>dim</i>	A pblong array to hold the indexes of all dimensions of the array. The size of the array must equal the dimensions of <i>array</i> .

Return value IPB_Value*.

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

```

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)`

Argument	Description
<i>value</i>	The value to be returned

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, pblob value, pbboolean IsNull )
AddBlobArgument ( PBCallInfo *ci, pblob value, pbboolean IsNull )
AddBoolArgument ( PBCallInfo *ci, pbboolean value, pbboolean IsNull )
AddByteArgument ( PBCallInfo *ci, pbyte value, pbboolean IsNull )
AddCharArgument ( PBCallInfo *ci, pbchar value, pbboolean IsNull )
AddDateArgument ( PBCallInfo *ci, pdate value, pbboolean IsNull )
AddDateTimeArgument ( PBCallInfo *ci, pdatetime value, pbboolean IsNull )
AddDecArgument ( PBCallInfo *ci, pbdec value, pbboolean IsNull )
AddDoubleArgument ( PBCallInfo *ci, pdouble value, pbboolean IsNull )
AddIntArgument ( PBCallInfo *ci, pint value, pbboolean IsNull )
AddLongArgument ( PBCallInfo *ci, plong value, pbboolean IsNull )
AddLongLongArgument ( PBCallInfo *ci, plonglong value, pbboolean IsNull )
AddObjectArgument ( PBCallInfo *ci, pobject value, pbboolean IsNull )
AddPBStringArgument ( PBCallInfo *ci, pbstring value, pbboolean IsNull )
AddRealArgument ( PBCallInfo *ci, preal value, pbboolean IsNull )
AddStringArgument ( PBCallInfo *ci, LPCTSTR value, pbboolean IsNull )
AddTimeArgument ( PBCallInfo *ci, ptime value, pbboolean IsNull )
AddUIntArgument ( PBCallInfo *ci, puint value, pbboolean IsNull )
AddUlongArgument ( PBCallInfo *ci, pulong value, pbboolean IsNull )
```

Argument	Description
<i>ci</i>	The PBCallInfo to which the argument is to be added.
<i>value</i>	The value to be added to the arguments array.
<i>IsNull</i>	Indicates whether the argument is null. The default is false.

Return value

PBXRESULT. PBX_OK on success.

Examples

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

```
long Cmy_pbni:: f_Retrieve(IPB_Session* session, pbint
retrieve_args, pbject 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;
}
```

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)`

Argument	Description
<i>obj</i>	A valid PowerBuilder object handle

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)`

Argument	Description
<i>obj</i>	A valid PowerBuilder object handle

Return value `pbclass` or null on error.

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

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

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

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)

Argument	Description
<i>rs</i>	A pointer to an IPB_ResultSetAccessor object

Return value pbobject.

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 PowerShell for the functions is shown after the C++ code:

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

void main(int argc, char* argv[])
{
    HINSTANCE hinst = LoadLibrary("pbvm115.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;

    pbobject obj = session->NewObject(cls);
    if (obj == NULL) return;

    pbmethodID mid = session->GetMethodID(cls, "f_ret",
        PBRT_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
pobject rs = ci.returnValue->GetObject();
IPB_ResultSetAccessor* rsa =
    session->GetResultSetAccessor(rs);

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

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

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

```

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 V115
SQLCA.DBMS = "ODBC"
SQLCA.AutoCommit = False
SQLCA.DBParm = &
    "ConnectString='DSN=EAS Demo DB
V115;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.sharedata(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

FindClass

Description Searches for a class with a given name within a given group.

Syntax FindClass(pbgroup *group*, LPCTSTR *name*)

Argument	Description
<i>group</i>	The handle of the group in which the class resides
<i>name</i>	The class name in lowercase

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:

```
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*)

Argument	Description
<i>group</i>	The handle of the group in which the class resides
<i>classID</i>	The class name in lowercase

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
<i>name</i>	The group name in lowercase
<i>type</i>	An enumerated type defined in pbgroup_type

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:

```
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
<i>cls</i>	pbclass containing the method.
<i>methodName</i>	The string name of the method in lowercase.
<i>rt</i>	Type of the method: PBRT_FUNCTION for function or PBRT_EVENT for event.
<i>readableSignature</i>	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:

```

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 `pbsig115` tool.

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

Table 7-3: FindMatchingFunction readable signature examples

Function prototype	Signature
<code>void test1()</code>	<code>" "</code>
<code>int test2()</code>	<code>" "</code>
<code>string test3(int a, double b)</code>	<code>"int, double"</code>
<code>datastore test4(powerobject a[], double b[2 to 10, 1 to 7])</code>	<code>"powerobject [], double[2 to 10, 1 to 7]"</code>
<code>int test5(readonly int a[10,20], ref long c[])</code>	<code>"readonly int [10,20], ref long[]"</code>

`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)

Argument	Description
<i>ci</i>	A pointer to the preallocated PBCallInfo structure

Return value None.

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

```
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 ( parray array, pblong dim[ ], pbboolean& IsNull )
GetBoolArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetByteArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetCharArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetDateArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetDateTimeArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetDecArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetDoubleArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetIntArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetLongArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetLongLongArrayItem ( parray array, pblonglong dim[ ], pbboolean& IsNull )
GetObjectArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
GetRealArrayItem ( parray array, pblong dim[ ], pbboolean& IsNull )
```

GetStringArrayItem (pbararray array, pblong dim[], pbboolean& IsNull)

GetTimeArrayItem (pbararray array, pblong dim[], pbboolean& IsNull)

GetUIntArrayItem (pbararray array, pblong dim[], pbboolean& IsNull)

GetUlongArrayItem (pbararray array, pblong dim[], pbboolean& IsNull)

Argument	Description
<i>array</i>	A valid pbararray structure
<i>dim</i>	The dimension of the array item to be obtained
<i>IsNull</i>	Indicates whether the array item is null

Return value The value of the array item.

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

```

pbject      pPObject = NULL;
pbboolean   bIsNull = 0;
pblong      dim[1];

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

```

See also

- GetArrayInfo
- GetArrayItemType
- GetArrayLength
- IsArrayItemNull
- NewBoundedObjectArray
- NewBoundedSimpleArray
- NewUnboundedObjectArray
- NewUnboundedSimpleArray
- ReleaseArrayInfo
- SetArrayItemToNull
- SetArrayItemValue
- Set<type>ArrayItem

Get<type>Field

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

Syntax

- GetArrayField (pbject obj, pbfieldID fid, pbboolean& isNull)
- GetBlobField (pbject obj, pbfieldID fid, pbboolean& isNull)
- GetBoolField (pbject obj, pbfieldID fid, pbboolean& isNull)
- GetByteField (pbject obj, pbfieldID fid, pbboolean& isNull)

GetCharField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetDateField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetDateTimeField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetDecField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetDoubleField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetIntField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetLongField(pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetLongLongField(pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetObjectField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetRealField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetStringField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetTimeField (pobject *obj*, pfieldID *fid*, pbint *value*)
 GetUIntField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)
 GetULongField (pobject *obj*, pfieldID *fid*, pbboolean& *isNull*)

Argument	Description
<i>obj</i>	The handle of the object whose field is to be accessed
<i>fid</i>	The field ID of the specified object
<i>isNull</i>	Indicates whether the field is null

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:

```

pbboolean isNull;
pbstring pstr =
    session->GetStringField(proxy, fid, isNull);
if (pstr != NULL)
{
    myclass = session->GetString(pstr);
    // process myclass
}
  
```

See also

- GetFieldID
- GetFieldType
- GetNumOfFields
- IsFieldArray
- IsFieldNull
- IsFieldObject
- SetFieldToNull

Set<type>Field

Get<type>GlobalVar

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

Syntax

```

GetArrayGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetBlobGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetBoolGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetByteGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetCharGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetDateGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetDateTimeGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetDecGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetDoubleGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetIntGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetLongGlobalVar( pbfieldID fid, pbboolean& isNull )
GetLongLongGlobalVar( pbfieldID fid, pbboolean& isNull )
GetObjectGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetRealGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetStringGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetTimeGlobalVar ( pbfieldID fid, pbint value )
GetUIntGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetUlongGlobalVar ( pbfieldID fid, pbboolean& isNull )

```

Argument	Description
<i>fid</i>	The field ID of the global variable
<i>isNull</i>	Indicates whether the variable is null

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:

```

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

```

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 )
  
```

Argument	Description
<i>group</i>	The group whose shared variable is to be accessed
<i>fid</i>	The field ID of the shared variable
<i>isNull</i>	Indicates whether the variable is null

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

Examples

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

```
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

GetSharedVarID
 GetSharedVarType
 IsSharedVarArray
 IsSharedVarNull
 IsSharedVarObject
 Set<type>SharedVar
 SetSharedVarToNull

GetArrayInfo

Description

Obtains information about an array.

Syntax

GetArrayInfo(pbararray *array*)

Argument	Description
<i>array</i>	A valid array handle

Return value

PBArrayInfo*.

Examples

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

```
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 PBArrayInfo is undetermined. The returned PBArrayInfo 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(parray *array*, pblong *dim*[])

Argument	Description
<i>array</i>	A valid parray structure.
<i>dim</i>	A pblong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of <i>array</i> .

Return value pbuint.

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

GetArrayLength

Description Obtains the length of an array.

Syntax `GetArrayLength(pbararray array)`

Argument	Description
<i>array</i>	A valid array handle

Return value `pblong`.

Examples This IF-ELSE statement populates a `PBArrayInfo` 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:

```
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(pblob *bin*)

Argument	Description
<i>bin</i>	A pointer to the source buffer

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:

```
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 (pblob *bin*)

Argument	Description
<i>bin</i>	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:

```
PBCallInfo* ci = new PBCallInfo;
pblob 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 (pbject *obj*)

Argument	Description
<i>obj</i>	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:

```

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();
        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)`

Argument	Description
<i>cls</i>	A valid class handle

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:

```

        LPCTSTR myClassName = session->GetClassName( myClass );
        size_t stLength = strlen( (LPCTSTR)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:

```

        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 pdate object to a string.

Syntax `GetDateString(pdate date)`

Argument	Description
<i>date</i>	The pdate data object to be converted to a string.

Return value LPCTSTR.

See also [NewDate](#)
[ReleaseDateString](#)
[SetDate](#)

GetDateTimeString

Description Converts data in a pdatetime object to a string.

Syntax `GetDateTimeString(pdatetime datetime)`

Argument	Description
<i>datetime</i>	The pdatetime data object to be converted to a string.

Return value LPCTSTR.

See also [NewDateTime](#)
[ReleaseDateTimeString](#)
[SetDateTime](#)

GetDecimalString

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

Syntax GetDecimalString(pbdec *dec*)

Argument	Description
<i>dec</i>	The pbdec data object to be converted to a string.

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:

```
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	<code>GetEnumItemValue(LPCTSTR <i>enumName</i>, LPCTSTR <i>enumItemName</i>)</code>
Return value	Long.
Examples	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"
```

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
----------	-----------------

GetException

Description	Obtains the current thrown exception object.
Syntax	GetException ()
Return value	pbobject.
Examples	This code gets the current exception object, clears the exception, and gets the class of the exception object:

```
pbclass cls;
pbobject 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 <i>cls</i> , LPCTSTR <i>fieldName</i>)

Argument	Description
<i>cls</i>	The class in which the field resides
<i>fieldName</i>	The instance member name, in lowercase

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:

```
void Callback::f_setvisible(IPB_Session* session,
    pbobject dwobj)
{
    pbclass cls;
    IPB_Value* pv;
    pbfieldID fid;
    pbstring strtmp;
    bool isTrue;
    pbboolean 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 GetFieldType
Get<type>Field
GetNumOfFields
IsFieldArray
IsFieldNull
IsFieldObject
Set<type>Field
SetFieldToNull

GetFieldName

Description Obtains the name of the specified field.

Syntax GetFieldName(pbclass *cls*, pbfieldID *fid*)

Argument	Description
<i>cls</i>	The class that defines the field
<i>fid</i>	The internal ID of the class instance variable

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, pbfieldID fid)`

Argument	Description
<i>cls</i>	The class that defines the field
<i>fid</i>	The internal ID of the class instance variable

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 82.

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

```
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)`

Argument	Description
<i>name</i>	The name of the global variable in lowercase

Return value `pbfieldID` 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:

```
fid = session -> GetGlobalVarID("l_gvar");
l_val = session -> GetLongGlobalVar(fid, isNull);
session -> SetLongGlobalVar(fid, l_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*)

Argument	Description
<i>fid</i>	The internal ID of the class instance variable

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*:

```

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

GetMarshaler

Description Obtains the marshaler object associated with a proxy object.

Syntax GetMarshaler(pbproxyObject *obj*)

Argument	Description
<i>obj</i>	An object of type pbproxyObject for which you want to find the marshaler.

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:

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

// 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)`

Argument	Description
<i>cls</i>	pbclass containing the function.
<i>methodName</i>	The string name of the method in lowercase.
<i>rt</i>	Type of the method: PBRT_FUNCTION for function or PBRT_EVENT for event.
<i>signature</i>	Internal signature of the PowerBuilder function, used to identify polymorphic methods in one class. Obtained with the pbsig115 tool. If the signature is a null string (" "), the first method found with the name <i>methodName</i> is returned.
<i>publicOnly</i>	A boolean that determines whether only public methods are searched (true) or all methods are searched (false). The default is true.

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:

```

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();
}

```

```

        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 37

GetMethodIDByEventID

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

Syntax GetMethodIDByEventID(pbclass *cls*, LPCTSTR *eventID*)

Argument	Description
<i>cls</i>	pbclass containing the method
<i>eventID</i>	A PowerBuilder predefined event string, such as pbm_bnclicked

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*:

```

    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
<i>obj</i>	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:

```
long f_retrieve(IPB_Session* session, puint iarg,
               pobject dwObj, pobject extObj)
{
    IMy_pbni* pMy_pbni = NULL;
    plong lRet;
    if (session -> IsNativeObject(extObj) )
    {
        pMy_pbni = (IMy_pbni*) session ->
            GetNativeInterface(extObj);

        lRet = pMy_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)`

Argument	Description
<i>cls</i>	A valid class handle for the class whose field is to be accessed

Return value `pblong`.

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

```
pbclass clz = d_session->GetClass(d_pobj);
pblong nf = d_session->GetNumOfFields(clz);
```

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

GetPBAnyArrayItem

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

Syntax `GetPBAnyArrayItem(pbarray array, pblong dim[], pbboolean& isNull)`

Argument	Description
<i>array</i>	A valid pbarray structure.
<i>dim</i>	A pblong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of <i>array</i> .
<i>isNull</i>	Indicates whether the variable is null

Return value `IPB_Value*`.

Usage See `GetPBAnyField`.

See also
 GetPBAnyField
 GetPBAnyGlobalVar
 GetPBAnySharedVar

GetPBAnyField

Description Obtains the value of a variable of type Any.

Syntax `GetPBAnyField(pbject obj, pbfieldID fid, pbboolean& isNull)`

Argument	Description
<i>obj</i>	A valid object handle for the object whose value is to be obtained
<i>fid</i>	The field ID of the variable
<i>isNull</i>	Indicates whether the variable is null

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:

```

session->PushLocalFrame();
pbgroup vgroup = session->FindGroup("n_test",
    pbgroup_userobject);
pbclass vcls = session->FindClass(vgroup, "n_test");
pbject 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:

```
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

GetPBAnyGlobalVar

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

Syntax `GetPBAnyGlobalVar(pbfieldID fid, pbboolean& isNull)`

Argument	Description
<i>fid</i>	The field ID of the variable
<i>isNull</i>	Indicates whether the variable is null

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)`

Argument	Description
<i>group</i>	The group to which the variable belongs
<i>fid</i>	The field ID of the variable
<i>isNull</i>	Indicates whether the variable is null

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
<i>name</i>	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 (pobject *rs*)

Argument	Description
<i>rs</i>	A pobject 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*:

```
pobject 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(pbggroup group, LPCTSTR fieldname)`

Argument	Description
<i>group</i>	The group to which the shared variable belongs
<i>fieldname</i>	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:

```
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 `Get<type>SharedVar`
`GetSharedVarType`
`IsSharedVarArray`
`IsSharedVarNull`
`IsSharedVarObject`
`Set<type>SharedVar`
`SetSharedVarToNull`

GetSharedVarType

Description Obtains the datatype of the specified shared variable.

Syntax `GetSharedVarType (pbgroup group, pbfieldID fid)`

Argument	Description
<i>group</i>	The group to which the shared variable belongs
<i>fid</i>	The internal field ID of the shared variable

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*)

Argument	Description
<i>string</i>	A pointer to a pbstring

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:

```
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*)

Argument	Description
<i>string</i>	The pbstring whose length is to be determined

Return value pblong.

Examples These statements set the value of a pblong variable to the length of a string:

```
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*)

Argument	Description
<i>cls</i>	A valid class handle for the descendent class

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:

```
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

GetSystemClass

Description Returns the first system class that the input class inherits from.

Syntax `GetSystemClass (pbclass cls)`

Argument	Description
<i>cls</i>	A descendent class whose ancestor system class is to be determined

Return value pbclass or null on error.

See also `GetMethodID`
`GetSystemGroup`

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:

```
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`

GetTimeString

Description Converts data in a pptime object to a string.

Syntax GetTimeString(pptime *time*)

Argument	Description
<i>time</i>	The pptime data object to be converted to a string.

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:

```
try
{
    session->InvokeObjectFunction(pboobj, mid, &ci);
    // Was PB exception thrown?
    if (session-> HasExceptionThrown())
    {
        // Handle PB exception
        session-> ClearException();
    }
}
```

See also ClearException
GetException
ThrowException

InitCallInfo

Description Initializes the PBCallInfo structure.

Syntax `InitCallInfo(pbclass cls, pbmethodID mid, PBCallInfo *ci)`

Argument	Description
<i>cls</i>	The pbclass containing the method
<i>mid</i>	The pbMethodID returned by GetMethodID
<i>ci</i>	A pointer to a preallocated PBCallInfo structure

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:

```
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)`

Argument	Description
<i>cls</i>	The class that contains the global function. If this is a system function, <i>cls</i> is obtained with <code>GetSystemFunctionsClass</code> ; otherwise, it is obtained with <code>FindGroup</code> and <code>FindClass</code> , with the function name as the group/class name.
<i>mid</i>	The <code>pbMethodID</code> returned by <code>GetMethodID</code> .
<i>ci</i>	A pointer to a preallocated <code>PBCallInfo</code> structure.

Return value PBXRESULT. Returns `PBX_OK` for success, or one of the following for failure:

```
PBX_E_INVALID_ARGUMENT
PBX_E_INVOKE_METHOD_INACCESSABLE
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:

```
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`

InvokeObjectFunction

Description Invokes a class member method.

Syntax `InvokeObjectFunction(pbobject obj, pbmethodID mid, PBCallInfo *ci)`

Argument	Description
<i>obj</i>	The pbject containing the method
<i>mid</i>	The pbMethodID returned by GetMethodID
<i>ci</i>	A pointer to a preallocated PBCallInfo structure

Return value PBXRESULT. Returns PBX_OK for success, or one of the following for failure:

```
PBX_E_INVALID_ARGUMENT
PBX_E_INVOKE_METHOD_INACCESSABLE
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:

```
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(pbararray *array*, pblong *dim*[])

Argument	Description
<i>array</i>	A valid pbararray structure that you want to check for a null-valued array item.
<i>dim</i>	A pblong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of <i>array</i> .

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)

Argument	Description
<i>cls</i>	A valid class handle or structure

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*)

Argument	Description
<i>cls</i>	A valid class handle for the class whose field is to be accessed
<i>fid</i>	The field ID of the specified object

Return value pbboolean.

Examples This code tests whether the field identified by *fid* is an array, and if so, gets the array value:

```

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 IsFieldNull(pboobject *obj*, pbfield *fid*)

Argument	Description
<i>obj</i>	A valid object handle for the object whose field is to be accessed
<i>fid</i>	The field ID of the specified object

Return value pbboolean.

Examples These statements test whether the field identified by *fid* is null:

```

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
<i>cls</i>	A valid class handle for the class whose field is to be accessed
<i>fid</i>	The field ID of the specified object

Return value pbboolean.

Examples These statements test whether the field identified by *fid* is an object:

```
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
<i>fid</i>	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:

```
fid = session -> GetGlobalVarID("arr_gvar");
if (session -> IsGlobalVarArray(fid))
{
    arr_val=session -> GetArrayGlobalVar(fid, isNull);
    ...
}
```

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*)

Argument	Description
<i>fid</i>	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:

```
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*)

Argument	Description
<i>fid</i>	The field ID of the global variable

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:

```
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<type>GlobalVar
- IsGlobalVarArray
- IsGlobalVarNull
- SetGlobalVarToNull
- Set<type>GlobalVar

IsNativeObject

Description Determines whether a pobject is an instance of a native class.

Syntax IsNativeObject(pobject *obj*)

Argument	Description
<i>obj</i>	A valid object handle

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:

```
long f_getrow(IPB_Session* session, pobject dwObj,
             pobject extObj)
{
    long lRet;
    IMy_pbni* pMy_pbni = NULL;
    IPBX_NonVisualObject* pp=NULL;

    if (session -> IsNativeObject(extObj) )
    {
        pp = (IPBX_NonVisualObject*) session ->
            GetNativeInterface(extObj);
        pMy_pbni = static_cast<IMy_pbni*>(pp);
        lRet = pMy_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*)

Argument	Description
<i>group</i>	The group whose shared variable is to be accessed
<i>fid</i>	The field ID of the shared variable

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*)

Argument	Description
<i>group</i>	The group whose shared variable is to be accessed
<i>fid</i>	The field ID of the shared variable

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)`

Argument	Description
<i>group</i>	The group whose shared variable is to be accessed
<i>fid</i>	The field ID of the shared variable

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)`

Argument	Description
<i>bin</i>	A void pointer that points to the source buffer
<i>len</i>	The length in bytes of the data in the buffer

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 `NewBoundedObjectArray(pbclass cls, pbuint dimension, PBAArrayInfo::ArrayBound* bounds)`

Argument	Description
<i>cls</i>	A valid class handle of the type of PowerBuilder object or structure array to be created
<i>dimension</i>	A number greater than one that indicates the dimension of the array to be created
<i>bounds</i>	An array containing the upper and lower boundaries of the array to be created

Return value pbarray or null on failure.

Examples

```
int size;
pbarray pbin_a;
PBAArrayInfo* ai;
PBXRESULT ret;
pbclass cls;
pbgroup group;

size = sizeof(PBAArrayInfo) +
        sizeof(PBAArrayInfo::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](#)

IsArrayItemNull
 NewBoundedSimpleArray
 NewUnboundedObjectArray
 NewUnboundedSimpleArray
 ReleaseArrayInfo
 Set<type>ArrayItem
 SetArrayItemToNull
 SetArrayItemValue

NewBoundedSimpleArray

Description Creates a bounded simple data array.

Syntax `NewBoundedSimpleArray(pbuint type, pbuint dimension, PBAArrayInfo::ArrayBound* bounds)`

Argument	Description
<i>type</i>	An enumerated variable of type <code>pbvalue_*</code> indicating the type of simple unbounded array to be created
<i>dimension</i>	A number greater than one that indicates the dimension of the array to be created
<i>bounds</i>	An array containing the upper and lower boundaries of the array to be created

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:

```
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:

```
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
    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

NewObject

Description Creates a new object of the specified type.

Syntax `NewObject(pbclass cls)`

Argument	Description
<i>cls</i>	The type of object or structure instance to be created

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

NewProxyObject

Description Creates a proxy for a remote object. The proxy is used to extend the network protocol in PowerBuilder.

Syntax `NewProxyObject(pbclass cls)`

Argument	Description
<i>cls</i>	The type of object or structure instance to be created

Return value pbproxyobject.

Examples

This example creates a new proxy object, creates a marshaler, and associates the marshaler with the proxy object:

```
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	<code>NewString(LPCTSTR)</code>
Return value	pbstring.
Examples	<pre> pbclass cls; cls = session->GetSystemFunctionsClass(); if(cls == NULL) { ret_val = session->NewString("null"); return ret_val; } </pre>
Usage	The returned string is destroyed when <code>PopLocalFrame</code> is called.
See also	SetString

NewTime

Description	Creates a new pbtime data object.
Syntax	NewTime()
Return value	pbtime.
Examples	<p>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:</p> <pre> Session->SplitTime(ci.returnValue->GetTime(), &hh, &mm, &ss); ret_val = Session-> NewTime(); Session-> SetTime(ret_val, hh, mm, ss); </pre>
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 <i>cls</i>)

Argument	Description
<i>cls</i>	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	Description
<i>type</i>	An enumerated variable of type <code>pbvalue_*</code> indicating the type of simple unbounded array to be created

Return value `pbararray` 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_*`. Most of the case statements have been removed for the sake of brevity:

```

if (d_returnType.isArray())
{
    returnValue.l = env->CallObjectMethodA(obj,
        mid, values.get());
    pbararray 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:

```
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:

```
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.

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:

```

if (pIPB_ObjectFactory)
{
    pIPB_ObjectFactory->Release();
    pIPB_ObjectFactory = NULL;
}
    
```

ReleaseArrayInfo

Description	Releases memory returned by GetArrayInfo.
Syntax	ReleaseArrayInfo(PBArrayInfo* <i>pbarrayinfo</i>)

Argument	Description
<i>pbarrayinfo</i>	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:

```

PBArrayInfo* ai;
...
session->ReleaseArrayInfo(ai);
    
```

Usage	If the array is an unbounded array, the bounds information in PBArraryInfo 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*)

Argument	Description
<i>string</i>	The string to be released from memory

Return value None.

See also GetDateString

ReleaseDateTimeString

Description Frees the memory acquired using GetDateTimeString.

Syntax ReleaseDateTimeString(LPCTSTR *string*)

Argument	Description
<i>string</i>	The string to be released from memory

Return value None.

See also GetDateTimeString

ReleaseDecimalString

Description Frees the memory acquired using GetDecimalString.

Syntax ReleaseDecimalString(LPCTSTR *string*)

Argument	Description
<i>string</i>	The string to be released from memory

Return value None.

See also GetDecimalString

ReleaseResultSetAccessor

Description Releases the pointer obtained using GetResultSetAccessor.

Syntax ReleaseResultSetAccessor (IPB_ResultSetAccessor* *rs*)

Argument	Description
<i>rs</i>	A pointer to the IPB_ResultSetAccessor object to be released

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*)

Argument	Description
<i>string</i>	The string to be released from memory

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:

```

pbstring psppls:: 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`

ReleaseTimeString

Description

Frees the memory acquired using `GetTimeString`.

Syntax

`ReleaseTimeString(LPCTSTR string)`

Argument	Description
<i>string</i>	The string to be released from memory

Return value

None.

See also

`GetTimeString`

ReleaseValue

Description Frees the IPB_Value acquired using AcquireValue or AcquireArrayItemValue.

Syntax ReleaseValue(IPB_Value* value)

Argument	Description
value	The string to be released from memory

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:

```
// 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 (pbject *obj*)

Argument	Description
<i>obj</i>	A valid PowerBuilder object handle

Return value None.

Examples

```
void MyPBNIClass::reference()
{
    d_session->AddGlobalRef(d_pobject);
}

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

See also AddGlobalRef

RemoveLocalRef

Description Removes a local reference to the specified PowerBuilder object.

Syntax RemoveLocalRef (pbject *obj*)

Argument	Description
<i>obj</i>	A valid PowerBuilder object handle

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*)

Argument	Description
<i>name</i>	The name of the variable to be removed

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:

```
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

Set<type>ArrayItem

Description Assigns a value to an array item of a specific type.

Syntax

```
SetBlobArrayItem ( pbarray array, pblong dim[ ], pblob value )
SetBoolArrayItem ( pbarray array, pblong dim[ ], pbboolean 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, pblonglong dim[ ], pblong value )
SetObjectArrayItem ( pbarray array, pblong dim[ ], pbobject obj )
SetPBStringArrayItem ( pbarray array, pblong dim[ ], pbstring value )
```


SetRealArrayItem (pbararray array, pblong dim[], poreal value)
 SetStringArrayItem (pbararray array, pblong dim[], LPCTSTR value)
 SetTimeArrayItem (pbararray array, pblong dim[], pptime value)
 SetUintArrayItem (pbararray array, pblong dim[], pbuint value)
 SetUlongArrayItem (pbararray array, pblong dim[], pblong value)

Argument	Description
<i>array</i>	A valid pbararray handle.
<i>dim</i>	A pblong array to hold indexes of each dimension. The number of dimensions must equal the number of dimensions of the array.
<i>value</i>	The new value of the array item.

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;
PBArrayInfo::ArrayBound bound;

bound.lowerBound = 1;
bound.upperBound = size;
d_pbararray = 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_pbararray, dim,
        pValue);

    delete [] cstr;
}
  
```

```
pbv.SetArray (d_pbararray) ;
```

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
- SetArrayItemValue

Set<type>Field

Description A set of methods that set a new value in an instance field of an object.

Syntax

- SetArrayField (pobject *obj*, pfieldID *fid*, pbararray *value*)
- SetBlobField (pobject *obj*, pfieldID *fid*, pblob *value*)
- SetBoolField (pobject *obj*, pfieldID *fid*, pbboolean *value*)
- SetByteField (pobject *obj*, pfieldID *fid*, pbyte *value*)
- SetCharField (pobject *obj*, pfieldID *fid*, pchar *value*)
- SetDateField (pobject *obj*, pfieldID *fid*, pdate *value*)
- SetDateTimeField (pobject *obj*, pfieldID *fid*, pdatetime *value*)
- SetDecField (pobject *obj*, pfieldID *fid*, pdec *value*)
- SetDoubleField (pobject *obj*, pfieldID *fid*, pdouble *value*)
- SetIntField (pobject *obj*, pfieldID *fid*, pint *value*)
- SetLongField (pobject *obj*, pfieldID *fid*, plong *value*)
- SetLongLongField (pobject *obj*, pfieldID *fid*, plonglong *value*)
- SetObjectField (pobject *obj*, pfieldID *fid*, pobject *value*)
- SetPBStringField (pobject *obj*, pfieldID *fid*, pstring *value*)
- SetRealField (pobject *obj*, pfieldID *fid*, preal *value*)
- SetStringField (pobject *obj*, pfieldID *fid*, LPCTSTR *value*)
- SetTimeField (pobject *obj*, pfieldID *fid*, ptime *value*)

SetUIntField (pobject *obj*, pbfieldID *fid*, pbuint *value*)

SetUlongField (pobject *obj*, pbfieldID *fid*, pbulong *value*)

Argument	Description
<i>obj</i>	The handle of the object whose field is to be accessed
<i>fid</i>	The field ID of the specified object
<i>value</i>	The value to be set

Return value PBX_RESULT.

Examples These statements set a new string value in a string field:

```
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<type>GlobalVar

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

Syntax
 SetArrayGlobalVar (pbfieldID *fid*, pbarray *value*)
 SetBlobGlobalVar (pbfieldID *fid*, pbblob *value*)
 SetBoolGlobalVar (pbfieldID *fid*, pbboolean *value*)
 SetByteGlobalVar (pbfieldID *fid*, pbbyte *value*)
 SetCharGlobalVar (pbfieldID *fid*, pbchar *value*)
 SetDateGlobalVar (pbfieldID *fid*, pbdate *value*)

SetDateTimeGlobalVar (pbfieldID *fid*, pbdatetime *value*)
 SetDecGlobalVar (pbfieldID *fid*, pbdec *value*)
 SetDoubleGlobalVar (pbfieldID *fid*, pbdouble *value*)
 SetIntGlobalVar (pbfieldID *fid*, pbint *value*)
 SetLongGlobalVar(pbfieldID *fid*, pblong *value*)
 SetLongLongGlobalVar(pbfieldID *fid*, pblonglong *value*)
 SetObjectGlobalVar (pbfieldID *fid*, pbobject *value*)
 SetPBStringGlobalVar (pbfieldID *fid*, pbstring *value*)
 SetRealGlobalVar (pbfieldID *fid*, poreal *value*)
 SetStringGlobalVar (pbfieldID *fid*, LPCTSTR *value*)
 SetTimeGlobalVar (pbfieldID *fid*, pbtime *value*)
 SetUIntGlobalVar (pbfieldID *fid*, pbuint *value*)
 SetULongGlobalVar (pbfieldID *fid*, pbulong *value*)

Argument	Description
<i>fid</i>	The field ID of the global variable
<i>value</i>	The value to be set

Return value

PBX_RESULT.

Examples

This code fragment shows how SetLongGlobalVar is used to add 1 to the value of a global variable:

```

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

See also

GetGlobalVarID
 GetGlobalVarType
 Get<type>GlobalVar
 IsGlobalVarObject
 SetGlobalVarToNull

Set<type>SharedVar

Description

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

Syntax

SetArraySharedVar (pbgroup *group*, pbfieldID *fid*, pbarray *value*)
 SetBlobSharedVar (pbgroup *group*, pbfieldID *fid*, pblob *value*)
 SetBoolSharedVar (pbgroup *group*, pbfieldID *fid*, pbboolean *value*)

SetByteSharedVar (pbgroup *group*, pbfieldID *fid*, pbbyte *value*)
 SetCharSharedVar (pbgroup *group*, pbfieldID *fid*, pbchar *value*)
 SetDateSharedVar (pbgroup *group*, pbfieldID *fid*, pbdate *value*)
 SetDateTimeSharedVar (pbgroup *group*, pbfieldID *fid*, pbdatetime *value*)
 SetDecSharedVar (pbgroup *group*, pbfieldID *fid*, pbdec *value*)
 SetDoubleSharedVar (pbgroup *group*, pbfieldID *fid*, pbdouble *value*)
 SetIntSharedVar (pbgroup *group*, pbfieldID *fid*, pbint *value*)
 SetLongSharedVar(pbgroup *group*, pbfieldID *fid*, pblong *value*)
 SetLongLongSharedVar(pbgroup *group*, pbfieldID *fid*, pblonglong *value*)
 SetObjectSharedVar (pbgroup *group*, pbfieldID *fid*, pbobject *value*)
 SetPBStringSharedVar (pbgroup *group*, pbfieldID *fid*, pbstring *value*)
 SetRealSharedVar (pbgroup *group*, pbfieldID *fid*, poreal *value*)
 SetStringSharedVar (pbgroup *group*, pbfieldID *fid*, LPCTSTR *value*)
 SetTimeSharedVar (pbgroup *group*, pbfieldID *fid*, pbtime *value*)
 SetUIntSharedVar (pbgroup *group*, pbfieldID *fid*, pbuint *value*)
 SetUlongSharedVar (pbgroup *group*, pbfieldID *fid*, pbulong *value*)

Argument	Description
<i>group</i>	The group whose shared variable is to be accessed
<i>fid</i>	The field ID of the shared variable
<i>value</i>	The value to be set

Return value

PBX_RESULT.

See also

Get<type>SharedVar
 GetSharedVarID
 GetSharedVarType
 IsSharedVarArray
 IsSharedVarNull
 IsSharedVarObject
 SetSharedVarToNull

SetArrayItemToNull

Description Sets the value of an array item to a null value.

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

Argument	Description
<i>array</i>	A valid pbarray structure in which you want to set an array item to null.
<i>dim</i>	A pblong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of <i>array</i> .

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*)

Argument	Description
<i>array</i>	A valid pbarray structure in which you want to set an array item to null.
<i>dim</i>	A pblong array to hold the indexes of each dimension of the array. The size of the array must equal the dimensions of <i>array</i> .
<i>src</i>	The value to which the array item is to be changed.

Return value None.

Examples This code sets the value of each item in an array:

```
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.

See also `AcquireArrayItemValue`
`ReleaseValue`
`SetArrayItemToNull`
`SetValue`

SetBlob

Description Destroys the existing data in a blob and copies data into it from a buffer.

Syntax `SetBlob (pbblob blob, const void* bin, pblong len)`

Argument	Description
<i>blob</i>	A valid pbblob object whose value is to be reset
<i>bin</i>	A pointer to the source buffer
<i>len</i>	The length in bytes of the data in the buffer

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, pblint year, pblint month, pblint day)`

Argument	Description
<i>date</i>	The pbdate object to be reset
<i>year</i>	A year in the range 1000 to 3000
<i>month</i>	A month in the range 1 to 12
<i>day</i>	A day in the range 1 to 31

Return value PBX_RESULT. 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);
```

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*, ppoint *year*, ppoint *month*, ppoint *day*, ppoint *hour*, ppoint *minute*, pdouble *second*)

Argument	Description
<i>dt</i>	The pbdatetime object to be reset
<i>year</i>	A year in the range 1000 to 3000
<i>month</i>	A month in the range 1 to 12
<i>day</i>	A day in the range 1 to 31
<i>hour</i>	An hour in the range 0 to 23
<i>minute</i>	A minute in the range 0 to 59
<i>second</i>	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:

```
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

Description Sets the value of a decimal variable to decimal data in a string.

Syntax SetDecimal(pbdec *dec*, LPCTSTR *dec_str*)

Argument	Description
<i>dec</i>	The decimal data object to be set
<i>dec_str</i>	The string containing the data to be converted to a decimal

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:

```
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)`

Argument	Description
<i>obj</i>	A valid object handle
<i>fid</i>	The field ID of the specified object

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*)

Argument	Description
<i>fid</i>	The field ID of the global variable

Return value None.

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

SetMarshaler

Description Sets a marshaller 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* *marshaller*)

Argument	Description
<i>obj</i>	An object of type pbproxyObject to be used as a proxy for a remote object that was created using NewProxyObject
<i>marshaller</i>	A class inherited from IPBX_Marshaler

Return value None.

Examples This example creates a JavaMarshaler class and associates it with a proxy object:

```
// Create JavaMarshaler
JavaMarshaler* marshaller = new JavaMarshaler(env,
    proxy, jobj);

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

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)`

Argument	Description
<i>name</i>	The name of the property to be set
<i>data</i>	A pointer to the data buffer where the variable's value resides

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);
}
```

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`:

```
if (mid == mid_GetProp)
{
    int* GetValue;
    GetValue = (int *)session -> GetProp(prop_name);
    ci -> returnValue -> SetInt(*GetValue);
}
```

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*)

Argument	Description
<i>group</i>	The group to which the shared variable belongs
<i>fid</i>	The field ID of the shared variable

Return value None.

Examples This example tests the IsSharedVarNull and SetSharedVarToNull functions:

```
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*)

Argument	Description
<i>string</i>	A valid pbstring variable whose value is to be replaced
<i>src</i>	The string to be assigned to <i>string</i>

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:

```

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 pptime object.

Syntax SetTime (pptime *time*, puint *hour*, puint *minute*, pdouble *second*)

Argument	Description
<i>time</i>	The pptime object to be reset
<i>hour</i>	An hour in the range 0 to 23
<i>minute</i>	A minute in the range 0 to 59
<i>second</i>	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 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:

```

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

SetValue

Description Sets the value of one IPB_Value object to the value of another IPB_Value object.

Syntax SetValue(IPB_Value* *dest*, IPB_Value* *src*)

Argument	Description
<i>dest</i>	The value to be replaced
<i>src</i>	The value to which <i>dest</i> is to be changed

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 pbddate object into a year, month, and day.

Syntax SplitDate (pbddate *date*, pbint **year*, pbint **month*, pbint **day*)

Argument	Description
<i>date</i>	The pbddate object to be split
<i>year</i>	A year in the range 1000 to 3000
<i>month</i>	A month in the range 1 to 12
<i>day</i>	A day in the range 1 to 31

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*)

Argument	Description
<i>dt</i>	The pbdatetime object to be split
<i>year</i>	A year in the range 1000 to 3000
<i>month</i>	A month in the range 1 to 12
<i>day</i>	A day in the range 1 to 31
<i>hour</i>	An hour in the range 0 to 23
<i>minute</i>	A minute in the range 0 to 59
<i>second</i>	A second in the range 0 to 59.999999

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(pbtime *time*, pbint **hour*, pbint **minute*, pbdouble **second*)

Argument	Description
<i>time</i>	The pbtime object to be split
<i>hour</i>	An hour in the range 0 to 23
<i>minute</i>	A minute in the range 0 to 59
<i>second</i>	A second in the range 0 to 59.999999

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:

```
Session->SplitTime(ci.returnValue->GetTime(), &hh,
    &mm, &ss);
ret_val = Session->NewTime();
Session->SetTime(ret_val, hh, mm, ss);
```

See also NewTime
SetTime

ThrowException

Description Throws a PowerBuilder exception or inherited exception, and replaces the existing exception if there is one.

Syntax ThrowException (pbject *ex*)

Argument	Description
<i>ex</i>	The exception to be thrown. The exception must first be created with NewObject.

Return value None.

Examples This code creates a new exception object in the class user_exception_psp, invokes its SetMessage function, and throws the exception:

```
pbclass cls;
pbmethodID mid;
pbject 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);

ci-> pArgs[0].SetPBString(session, "Test exception");

session -> InvokeObjectFunction(ex,mid,ci);
session -> ThrowException(ex);
if (!ThrowToPB)
    session -> ClearException();
session -> FreeCallInfo(ci);
delete ci;
return;

```

See also

ClearException
 GetException
 HasExceptionThrown

TriggerEvent

Description

Triggers a PowerBuilder event.

Syntax

TriggerEvent(pbject *obj*, pbmethodID *mid*, PBCallInfo **ci*)

Argument	Description
<i>obj</i>	The pbject containing the method
<i>mid</i>	The pbMethodID returned by GetMethodID
<i>ci</i>	A pointer to a preallocated PBCallInfo structure

Return value

PBXRESULT. Returns PBX_OK for success, or one of the following for failure:

PBX_E_INVALID_ARGUMENT
 PBX_E_INVOKE_METHOD_INACCESSABLE
 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(pboobject *obj*, pbfieldID *fid*)

Argument	Description
<i>obj</i>	The pboobject whose user interface property needs to be changed
<i>fid</i>	The field ID of the object

Return value

PBXRESULT. Returns success or failure.

Examples

This function changes the title of a DataWindow control:

```
void Callback::f_newtitle(IPB_Session* session,
pbstring str_val, pboobject 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 7-4: IPB_Value methods

Method	Description
Get<type>	Set of datatype-specific methods that return a pointer to the data in IPB_Value
GetClass	Returns the class handle of a PowerBuilder object
GetType	Returns the datatype of a single data item or array
IsArray	Returns true if the IPB_Value instance contains an array, otherwise returns false
IsByRef	Returns true if the IPB_Value instance is passed by reference
IsEnum	Returns true if the IPB_Value instance contains a null value, otherwise returns false
IsObject	Returns true if the IPB_Value instance contains an object or object array, otherwise returns false
SetToNull	Used to set the data contained in the IPB_Value instance to null so that data can be reset
Set<type>	Set of datatype-specific methods that set the value of the IPB_Value instance

Get<type>

Description	A set of datatype-specific methods that return a pointer to the data in IPB_Value.
Syntax	<code>GetArray ()</code> <code>GetBlob ()</code> <code>GetBool ()</code> <code>GetByte ()</code> <code>GetChar ()</code> <code>GetDate ()</code> <code>GetDateTime ()</code> <code>GetDecimal ()</code> <code>GetDouble ()</code> <code>GetInt ()</code> <code>GetLong ()</code> <code>GetLongLong ()</code> <code>GetObject ()</code> <code>GetReal ()</code> <code>GetString ()</code> <code>GetTime ()</code> <code>GetUInt ()</code> <code>GetUlong ()</code>
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: <pre>Session -> SplitDate (ci-> pArgs -> GetAt (0) -> GetDate () , &yy , &mm , &dd) ;</pre>
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	<code>Set<type></code>

GetClass

Description	Returns the class handle of a PowerBuilder object.
Syntax	<code>GetClass()</code>
Return value	pbclass or null on error.
Examples	<pre>pbclass clz = ci-> pArgs-> GetAt(i)-> GetClass ();</pre>
See also	<code>Get<type></code> <code>GetType</code> <code>Set<type></code>

GetType

Description	Returns the datatype of a single data item or array.
Syntax	<code>GetType()</code>
Return value	pbuint
Examples	<pre>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; ... </pre>
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	<code>Get<type></code> <code>GetClass</code> <code>Set<type></code>

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: <pre>if (ci->pArgs->GetAt (i) ->IsArray ()) { pArguments [i] .array_val = ci->pArgs->GetAt (i) ->GetArray (); continue; }</pre>
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: <pre>if (ci->pArgs->GetAt (i) ->IsByRef ()) ...</pre>
See also	IsArray IsEnum IsObject

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:

```

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: <pre>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; } ... </pre>
See also	IsArray IsByRef IsEnum

Set<type>

Description	Set of datatype-specific methods that set the value of the IPB_Value instance.
Syntax	SetArray (pbararray <i>array</i>) SetBlob(pblob <i>blob</i>) SetBool (pbboolean <i>boolean</i>) SetByte (pbbyte <i>byte</i>) SetChar (pbchar <i>char</i>) SetDate (pbdate <i>date</i>) SetDateTime(pbdatetime <i>datetime</i>) SetDecimal (pbdecimal <i>dec</i>) SetDouble (pbdouble <i>double</i>) SetInt (pbint <i>int</i>) SetLong(pblong <i>long</i>) SetLongLong(pblonglong <i>longlong</i>)

	SetObject (pbject <i>object</i>)
	SetPBString (pbstring <i>string</i>)
	SetReal(pbrreal <i>real</i>)
	SetString (LPCTSTR <i>string</i>)
	SetTime(pptime <i>time</i>)
	SetUint(pbuint <i>uint</i>)
	SetUlong (pbulong <i>ulong</i>)
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:

```

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);

// Call IPB_Value SetPBString method
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("");

```

```

// Call IPB_Session SetString method
Session -> SetString(ret_val, Session->GetString
                    (ci->returnValue->GetString()));
Session -> FreeCallInfo(ci);
delete ci;
return ret_val;

```

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:

```

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);
    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 <i>applicationName</i> , LPCTSTR* <i>libraryList</i> , pbuint <i>numLibs</i> , IPB_Session** <i>session</i>)

Argument	Description
<i>applicationName</i>	The name of the current application object in lowercase
<i>libraryList</i>	The library list of the PowerBuilder application that contains the objects and functions to be called
<i>numLibs</i>	The number of libraries in the library list
<i>session</i>	A pointer to IPB_Session*, which will return the current IPB_Session pointer after the call

Return value PBXRESULT. PBX_OK for success.

Examples This example creates an IPB_Session with the simple library list *mydemo.pbl*:

```
IPB_Session* session;
IPB_VM* vm = NULL;
fstream out;
ifstream in;
PBXRESULT ret;

HINSTANCE hinst=LoadLibrary("pbvm115.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)
```

Argument	Description
<i>applicationName</i>	The name of the application object to be run, in lowercase
<i>libraryList</i>	The library list of the application
<i>numLibs</i>	The number of libraries in the library list
<i>commandLine</i>	Parameters to be passed to the application object
<i>session</i>	A pointer to IPB_Session*, which will return the current IPB_Session pointer after the call

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("pbvm115.dll");

        P_PB_GetVM getvm = (P_PB_GetVM)
            GetProcAddress(hPBVMInst, "PB_GetVM");

        IPB_VM* vm = NULL;

        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 7-5: IPBX_Marshaler methods

Method	Description
Destroy	Destroys an instance of an object inherited from the IPBX_Marshaler structure
GetModuleHandle	Returns the handle of the PBX that contains the native class
InvokeRemoteMethod	Used in PowerBuilder marshaler native classes to call remote methods

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:

```
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;
pbulong 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
<i>session</i>	This IPB session
<i>obj</i>	The proxy object for the remote object
<i>methodDesc</i>	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"
<i>ci</i>	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:

```
#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
    );
};
```

```

    virtual pbulong   GetModuleHandle();
    virtual void Destroy();

};

```

The associated C++ implementation file contains code like this:

```

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: <pre> void MyPBNIClass::Destroy() { delete this; } </pre>
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

Description	Calls methods in PowerBuilder native classes.
Syntax	Invoke(IPB_Session * session, pobject obj, pbmethodID mid, PBCallInfo *ci)

Argument	Description
<i>session</i>	This IPB session
<i>obj</i>	The PowerBuilder extension object to be invoked
<i>mid</i>	The pbMethodID returned by GetMethodID
<i>ci</i>	The parameters and return value setting for the call

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:

```
PBXRESULT PBNIExt::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;

  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*)

Argument	Description
<i>dwExStyle</i>	The extended window style
<i>lpWindowName</i>	The window name
<i>dwStyle</i>	The window style
<i>x</i>	The horizontal position of the window
<i>y</i>	The vertical position of the window
<i>nWidth</i>	The window's width
<i>nHeight</i>	The window's height
<i>hWndParent</i>	The handle of the parent or owner window
<i>hInstance</i>	The handle of the application instance

Return value HWND.

Examples This is part of a visual extension example available on the Sybase Web site:

```
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 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`

GetEventID

Description 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)`

Argument	Description
<i>hWnd</i>	The handle of the parent window.
<i>iMsg</i>	The message sent to the parent.
<i>wParam</i>	<p>The word parameter of the message.</p> <p>For WM_COMMAND, the high-order word specifies:</p> <ul style="list-style-type: none"> • 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. <p>The low-order word specifies the identifier of the control, accelerator, or menu.</p> <p>For WM_NOTIFY, this parameter contains the identifier of the control sending the message.</p>
<i>lParam</i>	<p>The long parameter of the message.</p> <p>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.</p> <p>For WM_NOTIFY, this parameter contains a pointer to a structure.</p>

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.

```
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;
}

```

Usage	<p>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.</p> <p>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 <i>pbevtid.h</i> header file. The identifiers in <i>pbevtid.h</i> are associated with PowerBuilder event token names. For example, the identifier PB_BNCLICKED is associated with the token name pbm_bnclicked.</p> <p>If the message is not mapped to an event, GetEventID returns the value PB_NULL and the message is discarded.</p>
See also	<p>CreateControl GetWindowClassName</p>

GetWindowClassName

Description	Returns the name of the window.
Syntax	GetWindowClassName()
Return value	LPCTSTR.
Examples	<p>The string returned by GetWindowClassName is passed as an argument to the CreateControl method:</p> <pre> LPCTSTR CVisualExt::GetWindowClassName () { return s_className; } </pre>
Usage	The window must be registered before you call GetWindowClassName.
See also	<p>CreateControl GetEventID</p>

PBArrayInfo structure

Description PBArrayInfo is a C++ structure used to hold information about arrays.

Properties

Table 7-6: PBArrayInfo members

Member	Type	Description
ArrayBound	Local struct declaration	Structure of type pblong containing the boundaries (<i>upperBound</i> , <i>lowerBound</i>) of a dimension.
BoundedArray	Enum data	Used in <i>arrayType</i> to identify that the array is a bounded array.
UnboundedArray	Enum data	Used in <i>arrayType</i> to identify that the array is an unbounded array.
arrayType	Enum type	Used in <code>IPB_Session::GetArrayInfo</code> to identify the datatype of the array. Do not set this variable manually.
valueType	pbuint	The datatype of array items. Set it to <i>pbvalue_type</i> if it is a simple type, or <i>pbobject</i> if the item is a class or structure.
numDimensions	pbuint	Number of dimensions of the array. An unbounded array can have only one dimension. The lower bound is one.
bounds	ArrayBound[]	Array bounds declaration array, used in a bounded array.

PBCallInfo structure

Description PBCallInfo is a C++ structure used to hold arguments and return type information in function calls between PBNI and PowerBuilder.

Table 7-7: PBCallInfo members

Member	Type	Description
pArgs	IPB_Arguments*	Interface used to access arguments
returnValue	IPB_Value	Holds return data after the call
returnClass	pbclass	Holds return class after the call

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 7-8: PB_DateData members

Field	Description
<i>year</i>	A short identifying the year
<i>month</i>	A short identifying the month
<i>day</i>	A short identifying the day
<i>filler</i>	A short used for structure alignment only

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 7-9: PB_DateTimeData members

Field	Description
<i>date</i>	A PB_DateData structure identifying the date
<i>time</i>	A PB_TimeData structure identifying the time

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 7-10: PB_TimeData members

Field	Description
<i>hour</i>	A short identifying the hour
<i>minute</i>	A short identifying the minute
<i>second</i>	A short identifying the second
<i>filler</i>	A short used for structure alignment only

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

Field	Description
<i>x</i>	X coordinate of the visual control relative to its parent control (for example, the window that contains it).
<i>y</i>	Y coordinate of the visual control relative to its parent control.
<i>width</i>	Width of the visual control.
<i>height</i>	Height of the visual control.
<i>objectName</i>	The name of the visual object, for example: <code>uo_1</code> .
<i>tag</i>	Field to be used to pass any value at the user's discretion.
<i>enabled</i>	Whether the visual control is enabled. Possible values are <code>true</code> and <code>false</code> .
<i>visible</i>	Whether the visual control is visible. Possible values are <code>true</code> and <code>false</code> . In the development environment, PowerBuilder does not call the <code>PBX_DrawVisualObject</code> function if this field is set to <code>false</code> and the <code>Design>Show Invisibles</code> menu item is not selected.
<i>borderstyle</i>	Border style of the visual control. A value of the <code>pbborder_style</code> enumerated variable. Possible values are: <ul style="list-style-type: none"> • 0 – none • 1 – shadowbox • 2 – box • 5 – lowered • 6 – raised
<i>backColor</i>	Background color of the visual control. You can obtain the RGB value of the background color using the Windows API functions <code>GetRValue</code> , <code>GetGValue</code> , and <code>GetBValue</code> .

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
<i>dim</i>	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
<i>dim</i>	The dimension of the array item to be tested

Return value pbboolean.

See also GetAt
 SetAt
 SetToNull

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)
```

Argument	Description
<i>dim</i>	The dimension of the array item to be set
<i>v</i>	A ValueType defined in <i>pbtraits.h</i>
<i>string</i>	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 ValueType:

```
template < typename T, pbvalue_type I>
void ArrayCreator<T, I>::f_unbounded_simple_array(
    IPB_Session* session,
    ifstream in,
    fstream 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 << " ";
}
}
delete []iarg;
out << endl;
return;
}

```

See also

GetAt

SetToNull

Description

Sets the value of the specified array item to null.

Syntax

SetToNull(pblong *dim* [])

Argument	Description
<i>dim</i>	The dimension of the array item to be set

Return value

None.

See also

GetAt
IsNull
SetAt

PBBoundedArrayCreator template class

Description	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 <i>pbtraits.h</i> and are <code>pbint</code> , <code>pbuint</code> , <code>pbbyte</code> , <code>pblong</code> , <code>pblonglong</code> , <code>pbulong</code> , <code>pbboolean</code> , <code>pbreal</code> , <code>pbdouble</code> , <code>pbdec</code> , <code>pbdate</code> , <code>pbtime</code> , <code>pbdatetime</code> , <code>pbchar</code> , <code>pbblob</code> , and <code>pbstring</code> . The second version is used to create a bounded array of strings.
Methods	PBBoundedArrayCreator has two methods: GetArray SetAt

GetArray

Description	Obtains an array that has been created.
Syntax	GetArray()
Return value	<code>pbarray</code> .
Examples	This example sets up an array, reads in values, and then obtains the values in the array:

```

LPCTSTR *ostr_a;
char **sp;
int i;
pbarray out_array;
arrayBounds* bounds;
pbuint dim1, dim2, current_dim;
pblong itemcount1, itemcount2;
PBXRESULT ret;
PBArrInfo* ai;
pbstring *iarg, *oarg;
typedef PBBoundedArrayCreator<pbvalue_string>
    BoundedStringArrayCreator;

in >> dim1;
// allocate memory for pointer bounds
bounds = (arrayBounds*)malloc(dim1*sizeof
    (PBArrInfo::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*)

Argument	Description
<i>dim</i>	The dimension of the array item to be set
<i>v</i>	A ValueType defined in <i>pbtraits.h</i>
<i>string</i>	A string of type pbstring or LPCTSTR

Return value None.

Examples This example shows the use of `SetAt` in arrays of a type specified by a `ValueType`:

```
// 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:

```
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 <i>dim</i> [], ValueType <i>v</i>)

For string arrays:

```
SetAt(pblong dim[], LPCTSTR string)
SetAt(pblong dim[], pbstring string)
```

Argument	Description
<i>dim</i>	The dimension of the array item to be set
<i>v</i>	A ValueType defined in <i>pbtraits.h</i>
<i>string</i>	A string of type pbstring or LPCTSTR

Return value	None.
Examples	This method is included in the example for <code>GetArray</code> .
See also	<code>GetArray</code>

PBObjectArrayAccessor class

Description	The <code>PBObjectArrayAccessor</code> class is used to access the items in an object array.
Methods	<code>PBObjectArrayAccessor</code> has two methods: <code>GetAt</code> <code>SetAt</code>

GetAt

Description	Obtains the array item at the specified dimension.
Syntax	<code>GetAt(pblong <i>dim</i>[])</code>
Return value	<code>pbject</code> .

Argument	Description
<i>dim</i>	The dimension of the array item to be set

Examples This example shows the use of `GetAt` in an object array:

```
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

Description Sets the array item at the specified dimension.

Syntax SetAt(*pblong dim*[], *pbobject obj*)

Argument	Description
<i>dim</i>	The dimension of the array item to be set
<i>obj</i>	A valid object handle

Return value None.

Examples This example shows the use of SetAt in an object array:

```

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 <i>pbtraits.h</i> and are pbint, pbbyte, pbuint, pblong, pblonglong, pbulong, pbboolean, pbreal, pbdouble, pbdec, pbdate, pbtime, pbdatetime, pbchar, 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
<i>pos</i>	A pblong identifying a position in the array
<i>v</i>	A ValueType defined in <i>pbtraits.h</i>
<i>string</i>	A string of type pbstring or LPCTSTR

Return value

None.

Examples

This example shows the use of `SetAt` in arrays of a type specified by a `ValueType`:

```
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
<i>pos</i>	A pblong identifying a position in the array
<i>v</i>	A ValueType defined in <i>pbtraits.h</i>
<i>string</i>	A string of type pbstring or LPCTSTR

Return value None.

See also GetArray

Exported methods

Description 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.

Methods

Table 7-12: Methods that must be exported by all extensions

Method	Required
<code>PBX_CreateNonVisualObject</code>	When the extension contains nonvisual native classes
<code>PBX_CreateVisualObject</code>	When the extension contains visual native classes
<code>PBX_DrawVisualObject</code>	When you want to be able to draw a visual representation of the visual object in the PowerBuilder development environment
<code>PBX_GetDescription</code>	In all extensions
<code>PBX_InvokeGlobalFunction</code>	When the extension contains global functions
<code>PBX_Notify</code>	When you need to initialize and uninitialize a session

`PBX_CreateNonVisualObject`

Description Creates a new instance of a nonvisual PowerBuilder extension object.

Syntax `PBX_CreateNonVisualObject(IPB_Session* pbsession, pobject pbobj, LPCTSTR xtraName, IPBX_NonVisualObject **obj).`

Argument	Description
<i>pbsession</i>	This IPB session
<i>pbobj</i>	The name of a pobject corresponding to the PowerBuilder extension object to be created
<i>xtraname</i>	The name of the PowerBuilder native class in lowercase
<i>obj</i>	The PowerBuilder extension object to be created

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, pobject pbobj, LPCTSTR xtraName, IPBX_NonVisualObject **obj).`

Argument	Description
<i>pbsession</i>	This IPB session
<i>pbobj</i>	The name of a pobject corresponding to the PowerBuilder extension object to be created
<i>xtraname</i>	The name of the PowerBuilder native class in lowercase
<i>obj</i>	The PowerBuilder extension object to be created

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,
    pobject           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](#)

PBX_DrawVisualObject

Description Draws a visual object in the PowerBuilder development environment.

Syntax `PBX_DrawVisualObject(HDC hDC, LPCTSTR className, const PBX_DrawItemStruct& property).`

Argument	Description
<i>hDC</i>	A handle to the device context of the object
<i>className</i>	The name of the visual extension object to be drawn
<i>property</i>	A PBX_DrawItemStruct structure specifying the display properties of the object

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):

```
PBEXPORT PBXRESULT PBXCALL PBX_DrawVisualObject
(
HDC hDC,
LPCTSTR xtraName,
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( xtraName, "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

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:

```

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.

```

PBEXPORT 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 setttext(string txt)\n"
        "event testmouse pbm_mousemove \n"
        "end class\n"

        "globalfunctions\n"
        "function int funca(int a, int b)\n"
        "function int funcb(int a, int b)\n"
        "end globalfunctions\n"
    };

    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.

```
Desc ::=
    class_desc | globalfunc_desc | forward_desc | [Desc]*

class_desc ::=
    class className from parentClass newline
    [methods_desc]* end class newline

globalfunc_desc ::=
    globalfunctions newline [func_desc]* end globalfunctions

forward_desc ::=
    forward newline [forwardtype_desc]* end forward

forwardtype_desc ::=
    class className from parentClass newline

className ::=
    a PowerBuilder token (cannot duplicate an existing group name)

parentClass ::=
    any class inherited from NonVisualObject or UserObject

newline ::=
    a newline character

methods_desc ::=
    method_desc [methods_desc]*

method_desc ::=
    func_desc | sub_desc | event_desc

func_desc ::=
    function returnType funcName(args_desc) newline

returnType ::=
    pbType

pbType ::=
    any PowerBuilder type | previous declared PBNI class

funcName ::=
    a PowerBuilder token

args_desc ::=
    None | arg_desc, [args_desc]*

arg_desc ::=
    [ ref | readonly ] pbType argName [array_desc]
```



```

argName ::=
    a PowerBuilder token

array_desc ::=
    array declaration of PowerBuilder

sub_desc ::=
    subroutine subName(args_desc) newline

event_desc ::=
    event returnType eventName(args_desc) newline
    | event eventName pbevent_token newline

pbevent_token ::=
    string

```

This syntax for *event_desc* allows you to map a Windows message directly to a PowerBuilder event:

```
event eventName pbevent_token newline
```

For more information, see “Event processing in visual extensions” on page 32.

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
<i>pbsession</i>	This IPB session
<i>functionname</i>	The name of the global function
<i>ci</i>	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 (strcmp(functionName, "bitor") == 0)  
    {  
        ci->returnValue->SetInt(arg1 | arg2);  
    }else if (strcmp(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

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:

```
#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("pbvm115.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

About this chapter

This chapter describes two tools provided with the PBNI SDK:

- The `pbsig115` tool gets the internal signature of a PowerBuilder function from a PBL name.
- The `pbx2pbd115` tool generates a PBD from a PBX.

When you install PowerBuilder, these tools are installed in the *SDK* subdirectory of your PowerBuilder 11.5 directory and in *Shared\PowerBuilder*.

Contents

Topic	Page
<code>pbsig115</code>	243
<code>pbx2pbd115</code>	246

pbsig115

Description

The PowerBuilder function signature is the internal signature of a PowerBuilder function that is used to identify polymorphism functions in a class. The `pbsig115` 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 `pbsig115` 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

`pbsig115 pbl_name`

Examples

This command extracts function signatures from one of the PBLs in the Code Examples sample application:

```
pbsig115 pbexamw1.pbl
```

Here is some of the output from the previous command:

```
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_treepublic 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:

```
function integer of_get_all_sales_orders (Ref
    s_sales_order astr_order[], date adt_date, integer
    ai_direction)
```

For this function, the pbsig115 tool returns the following string. The first argument is an unbounded array of type s_sales_order and is passed by reference:

```
/* IRCs_sales_order.[]YI */
```


Usage

The pbsig115 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:

```
/* IRI */
```

PowerBuilder arrays PowerBuilder arrays are indicated with a pair of square brackets `[]` as a suffix. For bounded arrays, the brackets enclose the bounds.

```
/* 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 pbsig115 maps to datatypes and other entities.

Table 8-1: Return value and argument representation in pbsig115 output

Output	Datatype
[]	array
A	any
B	boolean
C	class
D	double
E	byte
F	real
G	basictype
H	character
I	integer
J	cursor
K	longlong
L	long
M	decimal
N	unsigned integer (uint)
O	blob

Output	Datatype
P	dbproc
Q	No type (subroutine)
S	string
T	time
U	unsigned long (ulong)
W	datetime
Y	date
Z	objhandle

The passing style is indicated by a prefix on the type.

Table 8-2: Passing style and varargs representation in pbsig115 output

Prefix	Meaning
None	Pass by value
R	Pass by reference
X	Pass as read only
V	Variable arguments (varargs)

pbx2pbd115

Description

The pbx2pbd115 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

```
pbx2pbd115 [+] 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:

```
pbx2pbd115 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.)

```
pbx2pbd115 + 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:

```
pbx2pbd115 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*:

```
pbx2pbd115 + 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 *pbx2pbd115* 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.

Appendix

This appendix describes wizards provided for Microsoft Visual Studio.

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

Topic	Page
Where the wizards are installed	251
Generating a PBNI project	253
Setting project options	253
Building and using the PBX	254

Where the wizards are installed

When you install PowerBuilder, the setup program installs four directories into the *PowerBuilder 11.5\SDK\PBNI\wizards* directory:

- *VCProjects 7.0*
- *VCProjects 7.1*
- *VCProjects 8.0*
- *VCWizards*

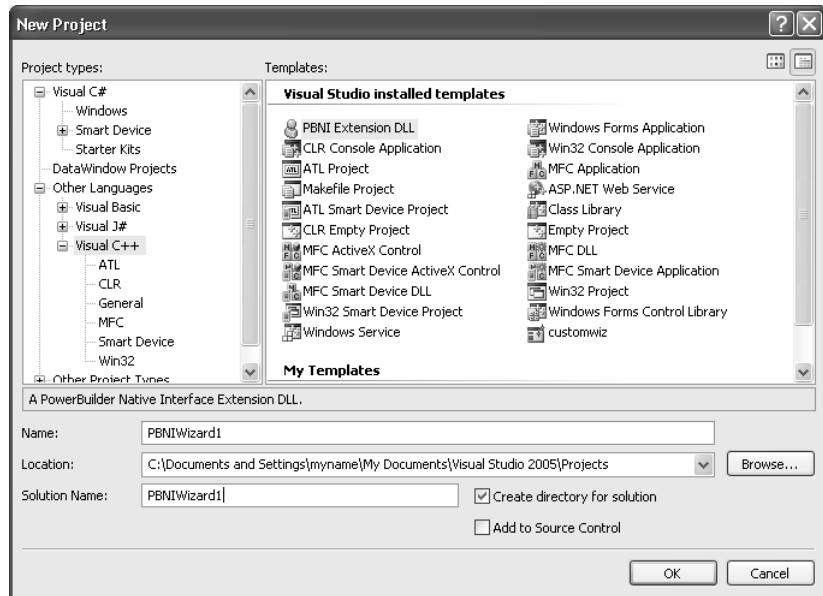
If Microsoft Visual Studio is already installed on your computer, the setup program also installs the appropriate files into your Visual Studio installation.

Table A-1: Where wizard files are installed

Visual Studio version	Files copied	Destination
Visual Studio 2005	VCProjects 8.0 VCWizards	..\Microsoft Visual Studio 8\VC\VCProjects ..\Microsoft Visual Studio 8\VC\VCWizards
Visual Studio .NET 2003	VCProjects 7.1 VCWizards	..\Microsoft Visual Studio .NET 2003\Vc7\VCProjects ..\Microsoft Visual Studio .NET 2003\Vc7\VCWizards
Visual Studio .NET 2002	VCProjects 7.0 VCWizards	..\Microsoft Visual Studio .NET\Vc7\VCProjects ..\Microsoft Visual Studio .NET\Vc7\VCWizards

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 11.5\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 26.

Index

A

- AcquireArrayItemValue function (IPB_Session) 100
- AcquireValue function (IPB_Session)
 - description 101
 - using 60
- AddArrayArgument function (IPB_Session) 102
- AddBlobArgument function (IPB_Session) 102
- AddBoolArgument function (IPB_Session) 102
- AddByteArgument function (IPB_Session) 102
- AddCharArgument function (IPB_Session) 102
- AddDateArgument function (IPB_Session) 102
- AddDateTimeArgument function (IPB_Session) 102
- AddDecArgument function (IPB_Session) 102
- AddDoubleArgument function (IPB_Session) 102
- AddGlobalRef function (IPB_Session) 104
- AddIntArgument function (IPB_Session) 102
- AddLocalRef function (IPB_Session) 104
- AddLongArgument function (IPB_Session) 102
- AddLongLongArgument function (IPB_Session) 102
- AddObjectArgument function (IPB_Session) 102
- AddPBStringArgument function (IPB_Session) 102
- AddRealArgument function (IPB_Session) 102
- AddRef function (IPB_ResultSetAccessor interface) 90, 93
- AddStringArgument function (IPB_Session) 102
- AddTimeArgument function (IPB_Session) 102
- AddUIntArgument function (IPB_Session) 102
- AddULongArgument function (IPB_Session) 102

C

- C++
 - calling PowerBuilder from 63
 - coding extensions 12
- calling PowerScript from an extension 37
- ClearException function (IPB_Session) 105
- code samples, on Web site 3

- conventions xii
- CreateControl function (IPBX_VisualObject) 210
 - using 31
- CreateResultSet function (IPB_Session) 105
- CreateSession function (IPB_VM) 201

D

- data
 - exchanging 53
 - saving values 59
- datatypes
 - for PowerBuilder data access 82
 - PowerBuilder to PBNI mapping 81
- debugging 41
- Destroy function (IPBX_Marshaler) 204
- Destroy function (IPBX_UserObject) 208

E

- enumerated types
 - pbgroup 82
 - pbroutine 83
 - pbvalue 83
 - support for 62, 125
- event IDs, and triggering events 35
- event processing, in visual extensions 32
- examples
 - calling PowerBuilder functions 40
 - nonvisual extension 11
 - on the Web 3
- exception handling 41
- exchanging data with PowerBuilder 53
- exported methods 230
- exporting methods 24
- extensions
 - creating 17, 27

Index

- marshaler, about 6
 - marshaler, creating 43
 - nonvisual, about 4
 - nonvisual, describing 19
 - nonvisual, example 11
 - using in PowerBuilder 25
 - visual, about 5
 - visual, creating 27
 - visual, creating instances 29
 - visual, event processing 32
 - visual, using 29
- F**
- FindClass function (IPB_Session) 109
 - FindClassByClassID function (IPB_Session) 109
 - FindGroup function (IPB_Session) 110
 - FindMatchingFunction function (IPB_Session) 110
 - using 39
 - forward declarations 20
 - FreeCallInfo function (IPB_Session) 112
- G**
- GetArray function (IPB_Value) 194
 - GetArray function (PB BoundedObjectArrayCreator class) 224
 - GetArray function (PB UnboundedArrayCreator template class) 227
 - GetArray function (PB UnboundedObjectArrayCreator class) 228
 - GetArray method (PB BoundedArrayCreator template class) 221
 - GetArrayField function (IPB_Session) 113
 - GetArrayGlobalVar function (IPB_Session) 115
 - GetArrayInfo function (IPB_Session) 117
 - GetArrayItemType (IPB_Session) 118
 - GetArrayLength function (IPB_Session) 119
 - GetArraySharedVar function (IPB_Session) 116, 140
 - GetAt (PB ArrayAccessor template class) 218
 - GetAt (PB ObjectArrayAccessor template class) 225
 - GetAt function (IPB_Arguments) 88
 - GetBlob function (IPB_Session) 120
 - GetBlob function (IPB_Value) 194
 - GetBlobArrayItem function (IPB_Session) 112
 - GetBlobField function (IPB_Session) 113
 - GetBlobGlobalVar function (IPB_Session) 115
 - GetBlobLength function (IPB_Session) 120
 - GetBlobSharedVar function (IPB_Session) 116, 140
 - GetBool function (IPB_Value) 194
 - GetBoolArrayItem function (IPB_Session) 112
 - GetBoolField function (IPB_Session) 113
 - GetBoolGlobalVar function (IPB_Session) 115
 - GetBoolSharedVar function (IPB_Session) 116, 140
 - GetByte function (IPB_Value) 194
 - GetByteArrayItem function (IPB_Session) 112
 - GetByteField function (IPB_Session) 113
 - GetByteGlobalVar function (IPB_Session) 115
 - GetByteSharedVar function (IPB_Session) 116
 - GetChar function (IPB_Value) 194
 - GetCharArrayItem function (IPB_Session) 112
 - GetCharField function (IPB_Session) 113
 - GetCharGlobalVar function (IPB_Session) 115
 - GetCharSharedVar function (IPB_Session) 116, 140
 - GetClass function (IPB_Session) 121, 195
 - GetClassName function (IPB_Session) 122
 - GetColumnCount function (IPB_ResultSetAccessor interface) 90
 - GetColumnMetaData function (IPB_ResultSetAccessor interface) 91
 - GetCount function (IPB_Arguments struct) 89
 - GetCurrGroup function (IPB_Session) 122
 - GetDate function (IPB_Value) 194
 - GetDateArrayItem function (IPB_Session) 112
 - GetDateField function (IPB_Session) 113
 - GetDateGlobalVar function (IPB_Session) 115
 - GetDateSharedVar function (IPB_Session) 116, 140
 - GetDateString function (IPB_Session) 123
 - GetDateTime function (IPB_Value) 194
 - GetDateTimeArrayItem function (IPB_Session) 112
 - GetDateTimeField function (IPB_Session) 113
 - GetDateTimeGlobalVar function (IPB_Session) 115
 - GetDateTimeSharedVar function (IPB_Session) 116, 140
 - GetDateTimeString function (IPB_Session) 123
 - GetDec function (IPB_Value) 194
 - GetDecArrayItem function (IPB_Session) 112
 - GetDecField function (IPB_Session) 113
 - GetDecGlobalVar function (IPB_Session) 115
 - GetDecimalString function (IPB_Session) 124

- GetDecSharedVar function (IPB_Session) 116, 140
- GetDouble function (IPB_Value) 194
- GetDoubleArrayItem function (IPB_Session) 112
- GetDoubleField function (IPB_Session) 113
- GetDoubleGlobalVar function (IPB_Session) 115
- GetDoubleSharedVar function (IPB_Session) 116, 140
- GetEnumItemName function (IPB_Session) 124
- GetEnumItemValue function (IPB_Session) 125
- GetEventID function (IPBX_VisualObject) 212
- GetException function (IPB_Session) 126
- GetFieldID function (IPB_Session) 126
- GetFieldName function (IPB_Session) 127
- GetFieldType function (IPB_Session) 128
- GetGlobalVarID function (IPB_Session) 128
- GetGlobalVarType function (IPB_Session) 129
- GetInt function (IPB_Value) 194
- GetIntArrayItem function (IPB_Session) 112
- GetIntField function (IPB_Session) 113
- GetIntGlobalVar function (IPB_Session) 115
- GetIntSharedVar function (IPB_Session) 116
- GetItemData function (IPB_ResultSetAccessor interface) 92
- GetLong function (IPB_Value) 194
- GetLongArrayItem function (IPB_Session) 112
- GetLongField function (IPB_Session) 113
- GetLongGlobalVar function (IPB_Session) 115, 116, 140
- GetLongLong function (IPB_Value) 194
- GetLongLongArrayItem function (IPB_Session) 112
- GetLongLongField function (IPB_Session) 113
- GetLongLongGlobalVar function (IPB_Session) 115
- GetMethodID function (IPB_Session) 131
- GetMethodID function (IPB_Session), about 37
- GetMethodIDByEventID function (IPB_Session) 132
- GetModuleHandle function (IPBX_Marshaler) 205
- GetNativeInterface function (IPB_Session) 133
- GetNumOfFields function (IPB_Session) 134
- GetObject function (IPB_Value) 194
- GetObjectArrayItem function (IPB_Session) 112
- GetObjectField function (IPB_Session) 113
- GetObjectGlobalVar function (IPB_Session) 115
- GetObjectSharedVar function (IPB_Session) 116, 140
- GetPBAAnyArrayItem function (IPB_Session) 134
- GetPBAAnyField function (IPB_Session) 135
- GetPBAAnyGlobalVar function (IPB_Session) 137
- GetPBAAnySharedVar function (IPB_Session) 137
- GetProp function (IPB_Session) 138
- GetReal function (IPB_Value) 194
- GetRealArrayItem function (IPB_Session) 112
- GetRealField function (IPB_Session) 113
- GetRealGlobalVar function (IPB_Session) 115
- GetRealSharedVar function (IPB_Session) 116, 140
- GetResultSetAccessor function (IPB_Session) 138
- GetRowCount function (IPB_ResultSetAccessor interface) 92
- GetSharedVarID function (IPB_Session) 139
- GetSharedVarType function (IPB_Session) 140
- GetString function (IPB_Session) 141
- GetString function (IPB_Value) 194
- GetStringArrayItem function (IPB_Session) 112
- GetStringField function (IPB_Session) 113
- GetStringGlobalVar function (IPB_Session) 115
- GetStringLength function (IPB_Session) 142
- GetStringSharedVar function (IPB_Session) 116
- GetSuperClass function (IPB_Session) 142
- GetSystemClass function (IPB_Session) 143
- GetSystemGroup function (IPB_Session) 143
- GetTime function (IPB_Value) 194
- GetTimeArrayItem function (IPB_Session) 112
- GetTimeField function (IPB_Session) 113
- GetTimeGlobalVar function (IPB_Session) 115
- GetTimeSharedVar function (IPB_Session) 116, 140
- GetTimeString function (IPB_Session) 144
- GetType function (IPB_Value) 195
- GetUint function (IPB_Value) 194
- GetUintArrayItem function (IPB_Session) 112
- GetUintField function (IPB_Session) 113
- GetUintGlobalVar function (IPB_Session) 115
- GetUintSharedVar function (IPB_Session) 116, 140
- GetUlong function (IPB_Value) 194
- GetUlongArrayItem function (IPB_Session) 112
- GetUlongField function (IPB_Session) 113
- GetUlongGlobalVar function (IPB_Session) 115
- GetUlongSharedVar function (IPB_Session) 116, 140
- GetWindowClassName function (IPBX_VisualObject) 214
- global functions 7
 - declaring 21

- describing 20
- exporting 20, 237
- implementing 23

H

- HasExceptionThrown function (IPB_Session) 144
- helper classes 7, 9

I

- InitCallInfo function (IPB_Session) 145
 - using 37
- interfaces
 - IPB_Arguments 88
 - IPB_Session 94
 - IPB_Value 193
 - IPB_VM 201
 - IPBX_Marshaler 204
 - IPBX_NonVisualObject 207
 - IPBX_UserObject 208
 - IPBX_VisualObject 210
 - overview 7
- Invoke function (IPBX_UserObject) 208
- InvokeClassFunction function (IPB_Session) 146
- InvokeObjectFunction function (IPB_Session) 147
- InvokeRemoteMethod function (IPBX_Marshaler) 206
- IPB_Arguments
 - about 54
 - description 88, 90, 93
 - using 54
- IPB_Session
 - about 57
 - interface description 94
 - list of functions 94
 - using 57
- IPB_Value
 - about 55
 - description 193
 - list of methods 193
 - saving data 59
 - using 55
- IPB_VM
 - about 6

- functions 201
- IPBX_Marshaler interface 204
- IPBX_NonVisualObject interface 207
- IPBX_NonVisualUserObject interface 207
- IPBX_UserObject interface 208
- IPBX_VisualObject interface 210
- IsArray function (IPB_Value) 196
- IsArrayItemNull function (IPB_Session) 148
- IsAutoInstantiate function (IPB_Session) 148
- IsByRef function (IPB_Value) 196
- IsEnum function (IPB_Value) 197
- IsFieldArray function (IPB_Session) 148
- IsFieldNull function (IPB_Session) 149
- IsFieldObject function (IPB_Session) 150
- IsGlobalVarArray function (IPB_Session) 150
- IsGlobalVarNull function (IPB_Session) 151
- IsGlobalVarObject function (IPB_Session) 152
- IsNull function (IPB_Value) 197
- IsNull function (PBArrayAccessor template class) 218
- IsObject function (IPB_Value) 198
- IsSharedVarArray function (IPB_Session) 154
- IsSharedVarNull function (IPB_Session) 154
- IsSharedVarObject function (IPB_Session) 155

J

- Java, calling from PowerBuilder 52
- JNI
 - calling extensions 52
 - compared with PBNI 10
 - used with marshaler extension 6
 - using with PBNI 76

M

- marshaler extensions
 - about 6
 - creating 43
 - describing 45
 - developing 44
 - generating proxies 51
 - implementing Creator class 46
 - implementing marshaler class 49
 - overview 43

method ID
 and GetMethodID 37
 getting 243

N

native class
 declaring 21
 describing 234
 implementing 22
 NewBlob function (IPB_Session) 155
 NewBoundedObjectArray function (IPB_Session)
 156
 NewBoundedSimpleArray function (IPB_Session)
 157
 NewDate function (IPB_Session) 158
 NewDateTime function (IPB_Session) 158
 NewDecimal function (IPB_Session) 159
 NewObject function (IPB_Session) 160
 NewProxyObject function (IPB_Session) 160
 NewString function (IPB_Session) 161
 NewTime function (IPB_Session) 162
 NewUnboundedObjectArray function (IPB_Session)
 162
 NewUnboundedSimpleArray function (IPB_Session)
 163
 nonvisual classes
 creating instances 24
 describing 19
 using 26
 nonvisual extensions
 about 4
 building 17
 describing 19
 example 11

P

passing values 53
 PB_DateData structure 216
 PB_DateTimeData structure 216
 PB_GetVM function (exported from PBVM) 240
 PB_TimeData structure 216

PBArrayAccessor
 in pbarry.cpp 9
 template class 218
 PBArrayInfo structure 215
 PBArrayInfoHolder 86
 PBBoundedArrayCreator
 in pbarry.cpp 9
 PBBoundedArrayCreator template class 221
 PBBoundedObjectArrayCreator
 in pbarry.cpp 9
 template class 224
 PBCallInfo structure
 reference 215
 saving data 59
 using 54
 PBD
 adding to library list 247
 generating 246, 247
 PBEventTrigger 86
 in pbfuniv.cpp 9
 pbext.awx 251
 PBGlobalFunctionInvoker 86
 in pbfuniv.cpp 9
 pbgroup enumerated types 82
 PBNI
 introduction 3
 Software Development Kit (SDK) 8
 PBOBJECTARRAYACCESSOR
 class 225
 in pbarry.cpp 9
 PBOBJECTCREATOR, in pbobject.cpp 9
 PBOBJECTFUNCTIONINVOKER 86
 in pbfuniv.cpp 9
 pboutine enumerated types 83
 pbsig110
 datatype mapping 245
 tool 243
 PBUnboundedArrayCreator
 in pbarry.cpp 9
 template class 227
 PBUnboundedObjectArrayCreator
 in pbarry.cpp 9
 template class 228
 pbvalue enumerated types 83
 PBVM
 embedding in a C++ application 6

Index

- interaction with extension 8
- loading 240
- PBX
 - building 25
 - importing 25
 - using instead of DLL as extension file type 11
- PBX_CreateNonVisualObject function 18, 230
- PBX_CreateVisualObject function 232
- PBX_DrawItemStruct structure 217
- PBX_DrawVisualObject function 233
- PBX_GetDescription function 18, 234
 - using 18
- PBX_GetVersion function 230
- PBX_InvokeGlobalFunction function 237
- PBX_Notify function 239
 - using with SetProperty 62
- pbx2pbd tool 246
- PBXResult error codes 84
- PBXRuntimeError exception 41
- PopLocalFrame function (IPB_Session) 164
- PowerBuilder extensions
 - building 11, 17, 25
 - calling PowerScript from 37
 - example 11
 - marshaler extensions 6, 43
 - nonvisual 4
 - overview 4
 - planning 18, 27
 - using 26
 - visual 5
- ProcessPBMessage function (IPB_Session) 164
- PushLocalFrame function (IPB_Session) 166

R

- ReferenceObject function (IPB_Session) 153
- RegisterClass, Windows method 30
- Release function (IPB_Session) 166
- ReleaseArrayInfo function (IPB_Session) 166
- ReleaseDateString function (IPB_Session) 167
- ReleaseDateTimeString function (IPB_Session) 167
- ReleaseDecimalString function (IPB_Session) 168
- ReleaseResultSetAccessor function (IPB_Session) 168
- ReleaseString function (IPB_Session) 168
- ReleaseTimeString function (IPB_Session) 169

- ReleaseValue function (IPB_Session) 170
- RemoveGlobalRef function (IPB_Session) 171
- RemoveLocalRef function (IPB_Session) 171
- RemoveProp function (IPB_Session) 172
- result sets, accessing 70
- RunApplication function (IPB_VM) 202

S

- samples, on the Web 3
- SDK, contents 8
- SetArray function (IPB_Value) 198
- SetArrayField function (IPB_Session) 174
- SetArrayGlobalVar function (IPB_Session) 175
- SetArrayItemToNull function (IPB_Session) 178
- SetArrayItemValue function (IPB_Session) 178
- SetArraySharedVar function (IPB_Session) 176
- SetAt function (PBArrayAccessor template class) 219
- SetAt function (PB BoundedArrayCreator template class) 222
- SetAt function (PB BoundedObjectArrayCreator template class) 224
- SetAt function (PBOBJECTArrayAccessor template class) 226
- SetAt function (PB UnboundedArrayCreator template class) 227
- SetAt function (PB UnboundedObjectArrayCreator class) 229
- SetBlob function (IPB_Session) 179
- SetBlob function (IPB_Value) 198
- SetBlobArrayItem function (IPB_Session) 172
- SetBlobField function (IPB_Session) 174
- SetBlobGlobalVar function (IPB_Session) 175
- SetBlobSharedVar function (IPB_Session) 176
- SetBool function (IPB_Value) 198
- SetBoolArrayItem function (IPB_Session) 172
- SetBoolField function (IPB_Session) 174
- SetBoolGlobalVar function (IPB_Session) 175
- SetBoolSharedVar function (IPB_Session) 176
- SetByte function (IPB_Value) 198
- SetByteArrayItem function (IPB_Session) 172
- SetByteField function (IPB_Session) 174
- SetByteGlobalVar function (IPB_Session) 175
- SetByteSharedVar function (IPB_Session) 176
- SetChar function (IPB_Value) 198

- SetCharArrayItem function (IPB_Session) 172
- SetCharField function (IPB_Session) 174
- SetCharGlobalVar function (IPB_Session) 175
- SetCharSharedVar function (IPB_Session) 176
- SetData function (IPB_RSItemData interface) 93
- SetDate function (IPB_Session) 179
- SetDate function (IPB_Value) 198
- SetDateArrayItem function (IPB_Session) 172
- SetDateField function (IPB_Session) 174
- SetDateGlobalVar function (IPB_Session) 175
- SetDateSharedVar function (IPB_Session) 176
- SetDateTime function (IPB_Session) 180
- SetDateTime function (IPB_Value) 198
- SetDateTimeArrayItem function (IPB_Session) 172
- SetDateTimeField function (IPB_Session) 174
- SetDateTimeGlobalVar function (IPB_Session) 175
- SetDateTimeSharedVar function (IPB_Session) 176
- SetDec function (IPB_Value) 198
- SetDecArrayItem function (IPB_Session) 172
- SetDecField function (IPB_Session) 174
- SetDecGlobalVar function (IPB_Session) 175
- SetDecimal function (IPB_Session) 180
- SetDecSharedVar function (IPB_Session) 176
- SetDouble function (IPB_Value) 198
- SetDoubleArrayItem function (IPB_Session) 172
- SetDoubleField function (IPB_Session) 174
- SetDoubleGlobalVar function (IPB_Session) 175
- SetDoubleSharedVar function (IPB_Session) 176
- SetFieldToNull function (IPB_Session) 181
- SetGlobalVarToNull function (IPB_Session) 182
- SetInt function (IPB_Value) 198
- SetIntArrayItem function (IPB_Session) 172
- SetIntField function (IPB_Session) 174
- SetIntGlobalVar function (IPB_Session) 175
- SetIntSharedVar function (IPB_Session) 176
- SetLong function (IPB_Value) 198
- SetLongArrayItem function (IPB_Session) 172
- SetLongField function (IPB_Session) 174
- SetLongGlobalVar function (IPB_Session) 175
- SetLongLong function (IPB_Value) 198
- SetLongLongGlobalVar function (IPB_Session) 175
- SetLongLongSharedVar function (IPB_Session) 176
- SetLongSharedVar function (IPB_Session) 176
- SetMarshaler function (IPB_Session) 130, 182
- SetNull function (IPB_RSItemData interface) 94
- SetNullValue function (IPB_Value) 200
- SetObject function (IPB_Value) 198
- SetObjectArrayItem function (IPB_Session) 172
- SetObjectField function (IPB_Session) 174
- SetObjectGlobalVar function (IPB_Session) 175
- SetObjectSharedVar function (IPB_Session) 176
- SetPBString function (IPB_Value) 198
- SetPBStringArrayItem function (IPB_Session) 172
- SetPBStringField function (IPB_Session) 174
- SetPBStringGlobalVar function (IPB_Session) 175
- SetPBStringSharedVar function (IPB_Session) 176
- SetProp function (IPB_Session) 183
- SetReal function (IPB_Value) 198
- SetRealArrayItem function (IPB_Session) 172
- SetRealField function (IPB_Session) 174
- SetRealGlobalVar function (IPB_Session) 175
- SetRealSharedVar function (IPB_Session) 176
- SetSharedVarToNull function (IPB_Session) 185
- SetString function (IPB_Session) 186
- SetString function (IPB_Value) 198
- SetStringArrayItem function (IPB_Session) 172
- SetStringField function (IPB_Session) 174
- SetStringGlobalVar function (IPB_Session) 175
- SetStringSharedVar function (IPB_Session) 176
- SetTime function (IPB_Session) 187
- SetTime function (IPB_Value) 198
- SetTimeArrayItem function (IPB_Session) 172
- SetTimeField function (IPB_Session) 174
- SetTimeGlobalVar function (IPB_Session) 175
- SetTimeSharedVar function (IPB_Session) 176
- SetToNull function (PBArrayAccessor template class) 220
- SetUInt function (IPB_Value) 198
- SetUIntArrayItem function (IPB_Session) 172
- SetUIntField function (IPB_Session) 174
- SetUIntGlobalVar function (IPB_Session) 175
- SetUIntSharedVar function (IPB_Session) 176
- SetUlong function (IPB_Value) 198
- SetUlongArrayItem function (IPB_Session) 172
- SetUlongField function (IPB_Session) 174
- SetUlongGlobalVar function (IPB_Session) 175
- SetUlongSharedVar function (IPB_Session) 176
- SetValue function (IPB_Session) 188
- Software Development Kit, contents 8
- SplitDate function (IPB_Session) 189
- SplitDateTime function (IPB_Session) 189
- SplitTime function (IPB_Session) 190

Index

system functions, calling 37

wizards, for Visual Studio 251

T

ThrowException function (IPB_Session) 190

TriggerEvent function (IPB_Session) 191

triggering events 35

typographical conventions xii

U

UpdateField function (IPB_Session) 192

V

variables, using throughout a session 61

Visual C++ wizard 251

visual classes

 creating instances 29

 declaring 28

 defining 27

 describing 19

 exporting methods 28

 implementing 28

 message processing 32

 registering 30

visual extensions

 about 5

 building 27

 creating instances 29

 event processing 32

 planning 27

 using 29

Visual Studio wizards 251

W

WindowProc, in visual extensions 32

Windows messages

 capturing 33

 processing 36

wizard, for Visual C++ 251