

ORCA Guide

Appeon PowerBuilder® 2019 R3
FOR WINDOWS

DOCUMENT ID: DC37664-01-1900-01

LAST REVISED: April 08, 2021

Copyright © Appeon. All rights reserved.

This publication pertains to Appeon 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.

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 Appeon Inc.

Appeon and other Appeon products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of Appeon Inc.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP and SAP affiliate company.

Java and all Java-based marks are trademarks or registered trademarks of Oracle and/or its affiliates 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.

Appeon Inc., 1/F, Shell Industrial Building, 12 Lee Chung Street, Chai Wan District, Hong Kong.

Contents

1 Using ORCA	1
1.1 What is ORCA?	1
1.1.1 What can ORCA do?	2
1.1.2 Who can develop programs that call ORCA?	2
1.2 Installing ORCA	3
1.3 ORCA and the Library painter	3
1.3.1 Objects in a PowerBuilder library	3
1.3.2 Object source code	3
1.3.3 PowerBuilder commands and ORCA functions	4
1.4 About ORCA functions	5
1.4.1 Functions for managing the ORCA session	5
1.4.2 Functions for managing PowerBuilder libraries	6
1.4.3 Functions for importing and compiling PowerBuilder objects	7
1.4.4 Functions for querying PowerBuilder objects	7
1.4.5 Functions for creating executables and dynamic libraries	8
1.4.6 Functions for deploying components to EAServer (Obsolete)	8
1.4.7 Functions for managing source control operations	9
1.5 About ORCA callback functions	9
1.5.1 ORCA functions that use callbacks	10
1.5.2 How a callback works	10
1.5.3 Content of a callback function	11
1.6 Writing ORCA programs	13
1.6.1 Outline of an ORCA program	13
1.6.1.1 First step: open a session	13
1.6.1.2 Optional step: set the library list and current application	14
1.6.1.3 Next steps: continuing with the ORCA session	15
1.6.1.4 Final step: close the session	15
1.6.2 Bootstrapping a new application	15
1.7 Removing obsolete ORCA functions	16
2 ORCA Functions	18
2.1 About the examples	18
2.2 ORCA return codes	18
2.3 PBORCA_ApplicationRebuild	19
2.4 PBORCA_BuildProject	21
2.5 PBORCA_BuildProjectEx	23
2.6 PBORCA_BuildProjectWithOverrides	24
2.7 PBORCA_CompileEntryImport	25
2.8 PBORCA_CompileEntryImportList	32
2.9 PBORCA_CompileEntryRegenerate	37
2.10 PBORCA_ConfigureSession	39
2.11 PBORCA_DeployWinFormProject	43
2.12 PBORCA_DynamicLibraryCreate	45
2.13 PBORCA_ExecutableCreate	47
2.14 PBORCA_LibraryCommentModify	53
2.15 PBORCA_LibraryCreate	54
2.16 PBORCA_LibraryDelete	55

2.17	PBORCA_LibraryDirectory	56
2.18	PBORCA_LibraryEntryCopy	59
2.19	PBORCA_LibraryEntryDelete	61
2.20	PBORCA_LibraryEntryExport	63
2.21	PBORCA_LibraryEntryExportEx	67
2.22	PBORCA_LibraryEntryInformation	69
2.23	PBORCA_LibraryEntryMove	72
2.24	PBORCA_ObjectQueryHierarchy	74
2.25	PBORCA_ObjectQueryReference	76
2.26	PBORCA_SccClose	78
2.27	PBORCA_SccConnect	78
2.28	PBORCA_SccConnectOffline	80
2.29	PBORCA_SccExcludeLibraryList	82
2.30	PBORCA_SccGetConnectProperties	83
2.31	PBORCA_SccGetLatestVersion	85
2.32	PBORCA_SccRefreshTarget	86
2.33	PBORCA_SccResetRevisionNumber	87
2.34	PBORCA_SccSetTarget	89
2.35	PBORCA_SessionClose	91
2.36	PBORCA_SessionGetError	92
2.37	PBORCA_SessionOpen	93
2.38	PBORCA_SessionSetCurrentAppl	93
2.39	PBORCA_SessionSetLibraryList	95
2.40	PBORCA_SetDebug	97
2.41	PBORCA_SetExeInfo	98
3	ORCA Callback Functions and Structures	101
3.1	Callback function for compiling objects	101
3.2	PBORCA_COMPERR structure	101
3.3	Callback function for deploying components to EAServer (Obsolete)	103
3.4	PBORCA_BLDERR structure	104
3.5	Callback function for PBORCA_LibraryDirectory	104
3.6	PBORCA_DIRENTRY structure	105
3.7	Callback function for PBORCA_ObjectQueryHierarchy	105
3.8	PBORCA_HIERARCHY structure	106
3.9	Callback function for PBORCA_ObjectQueryReference	106
3.10	PBORCA_REFERENCE structure	107
3.11	Callback function for PBORCA_ExecutableCreate	107
3.12	PBORCA_LINKERR structure	108
3.13	Callback function for PBORCA_SccSetTarget	108
3.14	PBORCA_SCCSETTARGET structure	109

1 Using ORCA

About this chapter

This chapter describes the Apeon Open Library API (ORCA).

It explains the correspondence between tasks a PowerBuilder developer can perform in the Library painter and tasks you want to do programmatically with ORCA for a PowerBuilder library.

It also explains the constraints involved in developing ORCA programs and who should and should not use ORCA, as well as the functions available in ORCA and how to conduct an ORCA session in your program.

1.1 What is ORCA?

ORCA is software for accessing the PowerBuilder Library Manager functions that PowerBuilder uses in the Library painter. A program (very often a C program) can use ORCA to do the same kinds of object and library management tasks that the Library painter interface provides.

History of ORCA

ORCA was created for CASE tool vendors as part of the CODE (Client/Server Open Development Environment) program. CASE tools needed programmatic access to PowerBuilder libraries to create and modify PowerBuilder objects based on an application design.

Typical ORCA programs

Applications use ORCA to manipulate PowerBuilder objects. They might:

- Write object source code and then use ORCA functions to place that object source in a PBL
- Extract objects from libraries using ORCA functions, modify the object source, and use ORCA again to put the objects back in the libraries

Sample ORCA applications

ORCA has been used for many types of tools that work with PowerBuilder, such as:

- OrcaScript utility
- CASE tools
- Class libraries
- Documentation tools
- Application management tools
- Utilities that might, for example, search for text and replace it throughout a library or display a tree view of objects in a library
- Interfaces for source control systems that PowerBuilder does not support directly

- Utilities to rebuild PowerBuilder targets from source-controlled objects

1.1.1 What can ORCA do?

ORCA lets your application do programmatically the same library and object management tasks that a developer performs in the PowerBuilder development environment. ORCA covers most of the functionality of the Library painter, and some of that of the Application and Project painters.

You can:

- Copy, delete, move, rename, and export objects in a PBL
- Import and compile objects
- Create an executable or a PowerBuilder Dynamic Library (PBD or DLL) with all of the options available in the Project painter
- Look at the ancestor hierarchy of an object or see which objects it references
- Create an entire application in a new library (called bootstrapping an application)
- Open PowerBuilder targets from source control and perform diverse source control operations on target objects

1.1.2 Who can develop programs that call ORCA?

ORCA as a development tool is designed for vendors who want to provide tools for PowerBuilder developers. Tool vendors must be aware of the constraints described in this section.

ORCA as a development tool is not meant for a wider audience of PowerBuilder developers. If you are a PowerBuilder developer, you should not develop programs that call ORCA unless you understand and observe the constraints described next.

Constraints when using ORCA

Both PowerBuilder and ORCA make use of the PowerBuilder compiler. However, the compiler is not reentrant, and multiple programs cannot use it simultaneously. Therefore, PowerBuilder cannot be running when your programs call ORCA.

Tool providers who use ORCA must code their programs carefully so that when a PowerBuilder developer calls their ORCA-based modules, their tool:

1. Exits PowerBuilder.
2. Performs the requested ORCA function.
3. Restarts PowerBuilder.

Caution

If the PowerBuilder development environment is not shut down while ORCA is running, your PowerBuilder libraries can become corrupted. For this reason, casual use of ORCA is not recommended.

1.2 Installing ORCA

ORCA is available to code partners, tool vendors, and customers who develop companion products and tools that manipulate and manage objects in PowerBuilder libraries for use with PowerBuilder.

To run ORCA programs

To run programs that use ORCA, you need the ORCA DLL (called PBORC.dll in PowerBuilder 2019 R3). When you install PowerBuilder, this DLL is installed in the same directory as other PowerBuilder DLLs.

To develop ORCA programs

To develop C programs that use ORCA, you need several items, available from the Appeon Developers Network website:

- C development files
 - PBORCA.H
 - PBORCA.LIB
- This documentation, available in PDF format

1.3 ORCA and the Library painter

A PowerBuilder library (PBL) is a binary file. It stores objects you define in the PowerBuilder painters in two forms: source and compiled. The source for an object is text. The compiled form is binary and is not readable by humans.

The Library painter lets the PowerBuilder developer view and maintain the contents of a PBL. The painter lists the objects in a PBL with their properties, such as modification date and comments.

In the Library painter, the PowerBuilder developer can delete, move, compile, export, and import objects, and can use source control systems and create PowerBuilder dynamic libraries and DLLs.

From the Library painter, you can open objects in their own painters and view and modify the objects graphically.

1.3.1 Objects in a PowerBuilder library

When you open an object in a painter, PowerBuilder interprets the library entries and displays the object in a graphical format. The painter does not display the source code. If you change the object graphically and save it again in the PBL, PowerBuilder rewrites the source code to incorporate the changes and recompiles the object.

1.3.2 Object source code

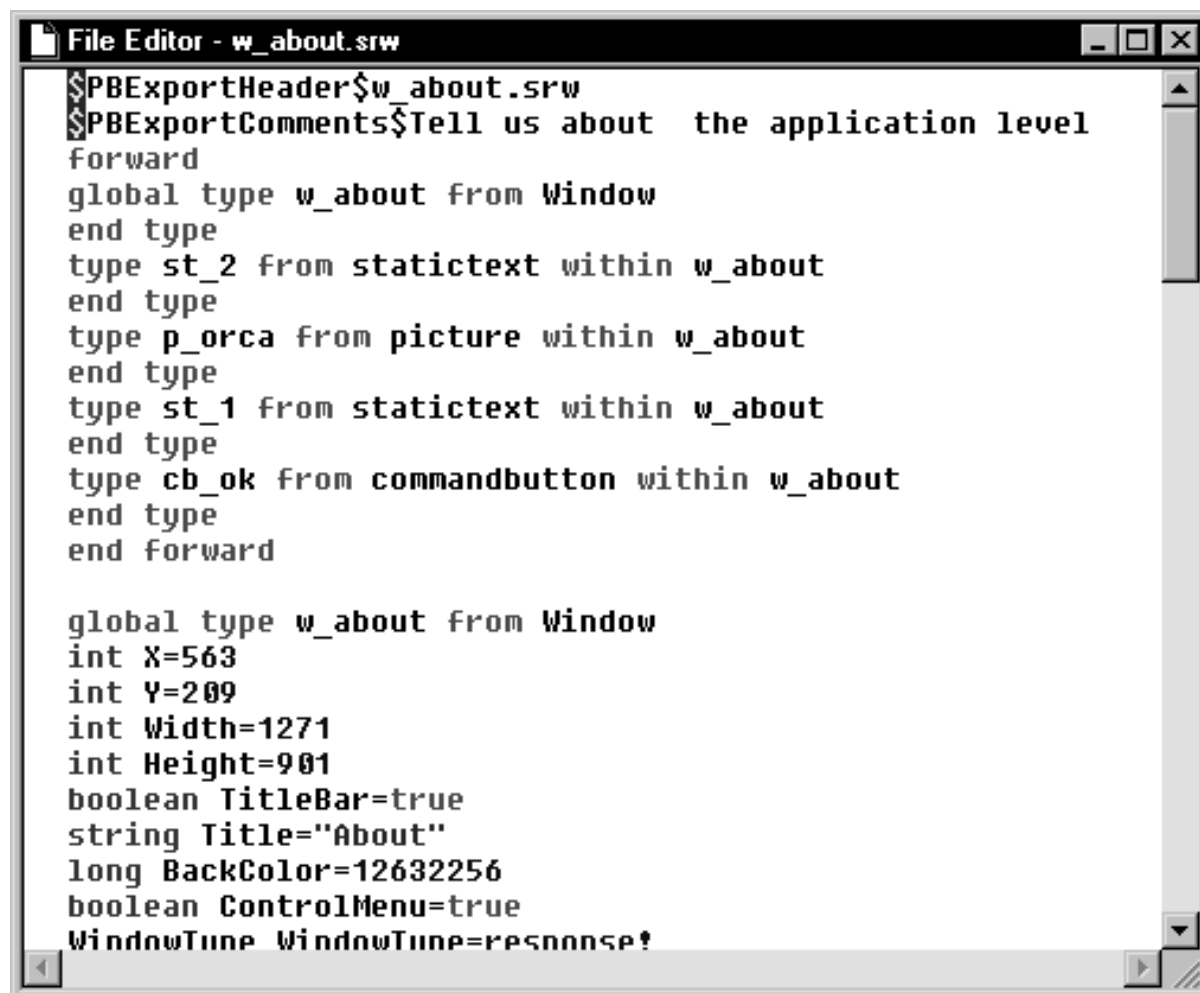
The Library painter lets you export source code, study and even modify it in any text editor, and import it back into the library. PowerBuilder compiles the imported object to check that the source code is valid. It will not import objects that fail to compile.

Source code exported to a file has two header lines before the source code:

```
$PBExportHeader$w_about.srw
$PBExportComments$Tell us about the application level
```

ORCA functions ignore these header lines and use the `lpszEntryName` and `lpszComments` arguments passed to the function.

You can view the exported source code in the PowerBuilder file editor:



```
File Editor - w_about.srw
$PBExportHeader$w_about.srw
$PBExportComments$Tell us about the application level
forward
global type w_about from Window
end type
type st_2 from statictext within w_about
end type
type p_orca from picture within w_about
end type
type st_1 from statictext within w_about
end type
type cb_ok from commandbutton within w_about
end type
end forward

global type w_about from Window
int X=563
int Y=209
int Width=1271
int Height=901
boolean TitleBar=true
string Title="About"
long BackColor=12632256
boolean ControlMenu=true
WindowTune WindowTune=response!
```

Learning source code syntax

The syntax for object source code is not documented. The only way to learn what belongs in source code is by exporting objects and studying their source.

ORCA and source code

ORCA has an export function so it can examine and modify existing objects. With PowerBuilder 10 and higher, a developer can configure the ORCA session to export source either to a memory buffer or to a file. The developer can also specify which of the four source encoding formats to use, whether or not to export the two export header lines, and whether or not to include the binary component of an object.

1.3.3 PowerBuilder commands and ORCA functions

Most ORCA functions have a counterpart in the Library painter, the Application painter, the Project painter, or the commands that start and stop a PowerBuilder session.

The next section identifies the ORCA functions, their purpose, and what they correspond to in the PowerBuilder development environment.

1.4 About ORCA functions

All ORCA functions are external C functions that use the WINAPI macro to specify the calling convention of the function. On the Windows platform, WINAPI is defined as `__stdcall`.

About the code examples in this book

All ORCA functions may be called from either an ANSI client program or a Unicode client program. The code examples in this book use macros that are defined in the `tchar.h` file that is installed with PowerBuilder in the `Shared/Appeon/PowerBuilder/cgen/h` directory. If the `/D _UNICODE` compiler directive is set, these macros accept Unicode string arguments. If `_UNICODE` is not defined, these macros accept ANSI string arguments. This coding technique allows you to create ORCA programs that run successfully as either ANSI or Unicode clients.

ORCA functions can be divided into seven groups with the following functions:

- Managing the ORCA session
- Managing PowerBuilder libraries
- Compiling PowerBuilder objects
- Querying PowerBuilder objects
- Creating executables and dynamic libraries
- Managing source control operations involving PowerBuilder objects

1.4.1 Functions for managing the ORCA session

Just as you begin a session in the PowerBuilder development environment by running PowerBuilder and end the session by exiting PowerBuilder, you need to open a session when using ORCA and close the session when finished.

Library list and current application

In the PowerBuilder development environment, you must first have a current application. You also set the library list search path if you plan to view or modify objects or create executables. ORCA has the same requirements, but in reverse order. In ORCA, you set the library list and then set the current application.

ORCA functions that do not involve compiling objects or building applications do not require a library list and current application. These are the library management functions. For source control functions, `PBORCA_SccSetTarget` implicitly sets the library list and current application.

Session management

Listed here are the session management functions (which all have the prefix `PBORCA_`), the purpose of each, and their equivalents in the PowerBuilder development environment:

Table 1.1:

Function (prefix <code>PBORCA_</code>)	Purpose	Equivalent in PowerBuilder
<code>ConfigureSession</code>	Sets session properties that affect the behavior of subsequent ORCA commands	Options
<code>SessionOpen</code>	Opens an ORCA session and returns the session handle	Starting PowerBuilder
<code>SessionClose</code>	Closes an ORCA session	Exiting PowerBuilder
<code>SessionSetLibraryList</code>	Specifies the libraries for the session	File>Library List
<code>SessionSetCurrentAppl</code>	Specifies the Application object for the session	File>Select Application
<code>SessionGetError</code>	Provides information about an error	No correspondence

1.4.2 Functions for managing PowerBuilder libraries

The library management functions are similar to commands in the Library painter. These functions allow you to create and delete libraries, modify library comments, and see the list of objects located within a library. They also allow you to examine objects within libraries; export their syntax; and copy, move, and delete entries.

These functions can be called outside the context of a library list and current application.

Listed here are the library management functions (which all have the prefix `PBORCA_`), the purpose of each, and their equivalents in the PowerBuilder Library painter:

Table 1.2:

Function (prefix <code>PBORCA_</code>)	Purpose	Equivalent in PowerBuilder
<code>LibraryCommentModify</code>	Modify the comments for a library	Library>Properties
<code>LibraryCreate</code>	Create a new library file	Library>Create
<code>LibraryDelete</code>	Delete a library file	Library>Delete
<code>LibraryDirectory</code>	Get the library comments and a list of its objects	List view
<code>LibraryEntryCopy</code>	Copy an object from one library to another	Entry>Copy
<code>LibraryEntryDelete</code>	Delete an object from a library	Entry>Delete

Function (prefix PBORCA_)	Purpose	Equivalent in PowerBuilder
LibraryEntryExport	Get the source code for an object	Entry>Export
LibraryEntryExportEx	Get the source code for an object	Entry>Export
LibraryEntryInformation	Get details about an object	List view
LibraryEntryMove	Move an object from one library to another	Entry>Move

1.4.3 Functions for importing and compiling PowerBuilder objects

These functions allow you to import new objects into a library from a text listing of their source code and to compile entries that already exist in a library.

Entries in a library have both a source code representation and a compiled version. When you import a new object, PowerBuilder compiles it. If there are errors, it is not imported.

You must set the library list and current application before calling these functions.

Listed here are the compilation functions (which all have the prefix PBORCA_), the purpose of each, and their equivalents in the PowerBuilder Library painter:

Table 1.3:

Function (prefix PBORCA_)	Purpose	Equivalent in Library painter
CompileEntryImport	Imports an object and compiles it	Entry>Import
CompileEntryImportList	Imports a list of objects and compiles them	No correspondence
CompileEntryRegenerate	Compiles an object	Entry>Regenerate
ApplicationRebuild	Compiles all the objects in all the libraries associated with an application	Design>Incremental Rebuild or Design>Full Rebuild

Compilation functions are not the functions that create an executable from a library. See [Functions for creating executables and dynamic libraries](#).

1.4.4 Functions for querying PowerBuilder objects

The object query functions get information about an object's ancestors and the objects it references.

You must set the library list and current application before calling these functions.

Listed here are the object query functions (which all have the prefix PBORCA_). There are no direct correspondences to PowerBuilder commands:

Table 1.4:

Function (prefix PBORCA_)	Purpose
ObjectQueryHierarchy	Gets a list of an object's ancestors
ObjectQueryReference	Gets a list of the objects an object refers to

1.4.5 Functions for creating executables and dynamic libraries

These functions allow you to create executables and PowerBuilder Dynamic Libraries (PBDs and DLLs). You can specify the same options for Pcode and machine code and tracing that you can specify in the Project painter.

Using ORCA, PBDs or DLLs must be created in a separate step before creating the executable.

You must set the library list and current application before calling these functions.

Listed here are the functions for creating executables and libraries (which all have the prefix PBORCA_), the purpose of each, and their equivalents in the PowerBuilder development environment:

Table 1.5:

Function (prefix PBORCA_)	Purpose	Equivalent in painter
ExecutableCreate	Creates an executable application using ORCA's library list and current Application object	Project painter
DynamicLibraryCreate	Creates a PowerBuilder dynamic library from a PBL	Project painter or Library painter: Library>Build Runtime Library
SetExeInfo	Sets additional file properties associated with the EXE and DLLs that are created	Project painter

1.4.6 Functions for deploying components to EAServer (Obsolete)

These functions are obsolete because EAServer is no longer supported since PowerBuilder 2017. An obsolete feature is no longer eligible for technical support and will no longer be enhanced, although it is still available.

These functions deploy an EAServer component using, or overwriting, specifications of the project object:

Table 1.6:

Function (prefix PBORCA_)	Purpose
BuildProject	Deploys component according to the project object specifications

Function (prefix PBORCA_)	Purpose
BuildProjectEx	Overrides server name and port number when deploying component

1.4.7 Functions for managing source control operations

These functions allow you to perform source control operations involving PowerBuilder targets and objects:

Table 1.7:

Function (prefix PBORCA_)	Purpose
ScClose	Closes the active SCC Project
ScConnect	Initializes source control and opens a project
ScConnectOffline	Simulates a connection to source control
ScExcludeLibraryList	Names the libraries in the target library list that you do not want to be synchronized in the next PBORCA_ScRefreshTarget operation
ScGetConnectProperties	Returns the SCC connection properties associated with a PowerBuilder workspace
ScGetLatestVersion	Copies the latest version of objects from the SCC repository to the local project path
ScRefreshTarget	Refreshes the source for each of the objects in target libraries
ScSetPassword	Sets the password property prior to ScConnect
ScSetTarget	Retrieves the target file from source control, passes the application object name to ORCA, and sets the ORCA session library list

1.5 About ORCA callback functions

Several ORCA functions require you to code a callback function. A callback function provides a way for the called program (the ORCA DLL or the Library Manager) to execute code in the calling program (the ORCA program executable).

How ORCA uses callbacks

ORCA uses callback functions when an unknown number of items needs to be processed. The purpose of the callback function is to process each of the returned items, and in most cases return the information to the user.

Optional or required

Some callbacks handle errors that occur when the main work is being done -- for example, when compiling objects or building executables. For handling errors, the callback function is optional. Other callbacks handle the information you wanted when you called the function -- such as each item in a directory listing. Callbacks for information functions are required.

Language requirement

ORCA functions that require the use of callback functions can be used only by programs written in languages that use pointers, such as C and C++.

When you create a new ORCA callback function, use the `CALLBACK` macro to specify the calling convention of the function. On the Windows platform, `CALLBACK` is defined as `__stdcall`.

1.5.1 ORCA functions that use callbacks

These functions (which all have the prefix `PBORCA_`) use a callback function:

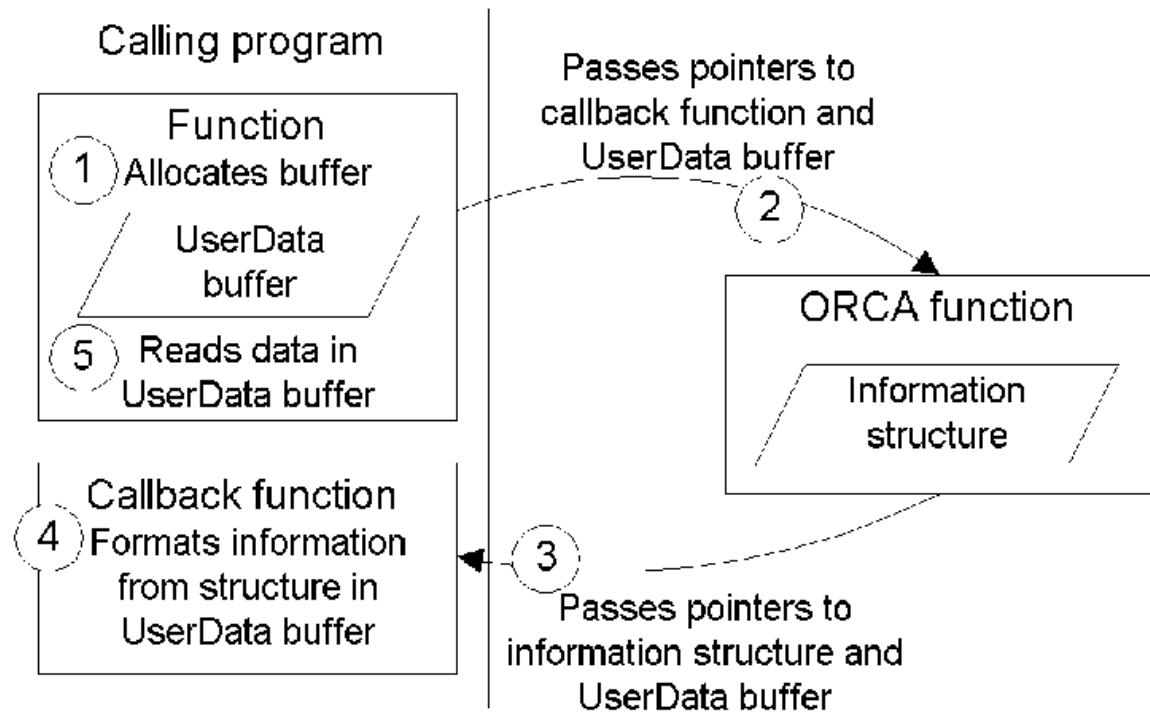
Table 1.8:

ORCA function call (prefix <code>PBORCA_</code>)	Purpose of callback
<code>BuildProjectEx</code>	Called once for each deployment error
<code>BuildProject</code>	
<code>CompileEntryImport</code>	Called once for each compile error
<code>CompileEntryImportList</code>	
<code>CompileEntryRegenerate</code>	
<code>ExecutableCreate</code>	Called once for each link error
<code>LibraryDirectory</code>	Called once for each library entry name
<code>ObjectQueryHierarchy</code>	Called once for every ancestor name
<code>ObjectQueryReference</code>	Called once for every object referenced in the entry
<code>SccSetTarget</code>	Called once for each library in the library list

1.5.2 How a callback works

ORCA calls a callback function like this:

1. The calling program allocates a buffer to hold data (the `UserData` buffer).
2. The calling program calls an ORCA function, passing it pointers to the callback function and the `UserData` buffer.
3. When the ORCA function needs to report information, it calls the callback function. It passes pointers to the structure holding the information and the `UserData` buffer.
4. The callback function reads the information in the structure and formats it in the `UserData` buffer.
Steps 3 and 4 repeat for each piece of information ORCA needs to report. An ORCA function might call the callback once, several times, or not at all, depending on whether errors occur or information needs to be reported.
5. The ORCA function completes and returns control to the calling program, which reads the information in the `UserData` buffer.



1.5.3 Content of a callback function

The processing that occurs in the callback function is entirely up to you. This section illustrates a simple way of handling it.

UserData buffer

In this example, the **UserData** buffer is a structure with a field whose value points to the actual message buffer. Other fields keep track of the message buffer's contents as it is filled:

```
typedef struct ORCA_UserDataInfo {
    LPBYTE lpszBuffer;    // Buffer to store data
    DWORD dwCallCount;   // # of messages in buffer
    DWORD dwBufferSize; // size of buffer
    DWORD dwBufferOffset; // current offset in buffer
} ORCA_USERDATAINFO, FAR *PORCA_USERDATAINFO;
```

Calling program

In the calling program, the **UserDataInfo** structure is initialized.

The calling program does not know how much room will be required for messages, so it allocates 60000 bytes (an arbitrary size). If you are gathering link errors, it's probably enough. It might not be enough if you wanted directory information for a large library:

```
ORCA_USERDATAINFO UserDataBuffer;
PORCA_USERDATAINFO lpUserDataBuffer;

lpUserDataBuffer = &UserDataBuffer;
lpUserDataBuffer->dwCallCount = 0;
lpUserDataBuffer->dwBufferOffset = 0;
lpUserDataBuffer->dwBufferSize = 60000;
lpUserDataBuffer->lpszBuffer =
    (LPTSTR)malloc((size_t)lpUserDataBuffer->
        dwBufferSize);
```

```
memset(lpUserDataBuffer->lpszBuffer,
       0x00, (size_t)lpUserDataBuffer->dwBufferSize);
```

Define function pointer

The calling program defines a function pointer to the callback function that it passes to the ORCA function:

```
PBORCA_LINKPROC fpLinkProc;
fpLinkProc = (PBORCA_LINKPROC)LinkErrors;
```

Call ORCA

The calling program calls the ORCA function, passing the callback function pointer and the UserData buffer pointer. This example calls `PBORCA_ExecutableCreate`, whose callback type is `PBORCA_LNKPROC`:

```
rtn = PBORCA_ExecutableCreate(..., (PBORCA_LNKPROC)
fpLinkProc, lpUserDataBuffer);
```

Process results

Finally, the calling program can process or display information that the callback function stored in the UserData buffer.

Free allocated memory

If your UserData structure allocates memory, free the allocated memory:

```
free( lpUserDataBuffer->lpszBuffer )
```

Callback program

The callback program receives a structure with the current error or information and stores the information in the message buffer pointed to by `lpszBuffer` in the UserData buffer. It also manages the pointers stored in the UserData buffer.

Simple callback

A simple callback might do the following:

- Keep count of the number of times it is called
- Store messages and reallocate buffer if it overflows

This code implements a callback called `LinkErrors` for `PBORCA_ExecutableCreate`:

```
void CALLBACK LinkErrors(PPBORCA_LINKERR lpLinkError,
                        LPVOID lpUserData)
{
    PORCA_USERDATAINFO lpData;
    LPBYTE lpCurrByte;
    LPTSTR lpCurrentPtr;
    int iNeededSize;
    lpData = (PORCA_USERDATAINFO) lpUserData;

    // Keep track of number of link errors
    lpData->dwCallCount++;

    // Is buffer already full?
    if (lpData->dwBufferOffset==lpData->dwBufferSize)
        return;
```



```

// How long is the new message?
// Message length plus carriage rtn and newline
iNeededSize =
    (_tcslen(lpLinkError->lpszMessageText) + 2)*
    sizeof(TCHAR);
// Reallocate buffer if necessary
if ((lpData->dwBufferOffset + iNeededSize) >
    lpData->dwBufferSize)
{
    LPVOID lpNewBlock;
    DWORD dwNewSize;
    dwNewSize = lpData->dwBufferSize * 2;
    lpNewBlock = realloc(lpData->lpszBuffer,
        (size_t)dwNewSize);
    if (lpNewBlock)
    {
        lpData->lpszBuffer = (LPTSTR) lpNewBlock;
        lpData->dwBufferSize = dwNewSize;
    }
    else
        return;
}

// Set pointer for copying message to buffer
lpCurrentPtr = lpData->lpszBuffer
    + lpData->dwBufferOffset;
lpCurrString = (LPTSTR) lpCurrByte;

// Copy link error message, CR, and LF to buffer.
_tcsncpy(lpCurrentPtr, lpLinkError->lpszMessageText);
_tcsncat(lpCurrentPtr, _TEXT("\r\n"));
lpData->dwBufferOffset += iNeededSize;
return;
}

```

1.6 Writing ORCA programs

This section outlines the skeleton of an ORCA program, beginning with opening a session. It also describes how to build an application from scratch without having to start with a library containing an Application object.

1.6.1 Outline of an ORCA program

To use the ORCA interface, your calling program will:

1. Open an ORCA session.
2. (Optional, depending on which ORCA functions you want to call.)
Set the library list and the current Application object.
3. Call other ORCA functions as needed.
4. Close the ORCA session.

1.6.1.1 First step: open a session

Before calling any other ORCA functions, you need to open a session. The `PBORCA_SessionOpen` function returns a handle that ORCA uses to manage this program's

ORCA session. The handle type HPBORCA is defined as LPVOID, meaning that it can be a pointer to any type of data. This is because within ORCA it is mapped to a structure not available to the calling program.

Sample code

This sample C function opens an ORCA session:

```
HPBORCA WINAPI SessionOpen()
{
    HPBORCA hORCASession;
    hORCASession = PBORCA_SessionOpen();
    return hORCASession;
}
```

1.6.1.2 Optional step: set the library list and current application

The next step in writing an ORCA program depends on the intent of the program. The choices are:

- If the program only manages libraries, moves entries among libraries, or looks at the source for entries, there are no other required calls. You can continue with your ORCA session.
- If the program calls other ORCA functions, you must set the library list and then set the current application.

Comparison to PowerBuilder

This is similar to the requirements of the PowerBuilder development environment. In the Library painter, you can copy entries from one PBL to another, even if they are outside the current application or library list. You can export the syntax of a library entry that is not in the library list. However, you can only import entries into libraries in the current application's library list.

In the PowerBuilder development environment, you select an Application object in the Application painter and then set the library search path on the Application object's property sheet. With ORCA, you set the library list first and then set the Application object.

Set once per session

You can set the library list and current application only once in an ORCA session. To use another library list and application, close the ORCA session and open a new session.

Sample code

This sample C function sets the library list and the current application:

```
int WINAPI SetUpSession(HPBORCA hORCASession)
{
    TCHAR szApplName[36];
    int nReturnCode;
    LPTSTR lpLibraryNames[2] =
        { _TEXT("c:\\pbfiles\\demo\\master.pbl"),
          _TEXT("c:\\pbfiles\\demo\\work.pbl") };

    // Call the ORCA function
    nReturnCode = PBORCA_SessionSetLibraryList(
        hORCASession, lpLibraryNames, 2);
    if (nReturnCode != 0)
```

```
return nReturnCode; // return if it failed

// Set up the string containing the appl name
_tcscpy(szApplName, _TEXT("demo"));

// The appl object is in the first library
nReturnCode = PBORCA_SessionSetCurrentAppl(
    hORCASession, lpLibraryName[0], szApplName)
return nReturnCode;
}
```

1.6.1.3 Next steps: continuing with the ORCA session

After the library list and application are set, you can call any ORCA function using the handle returned by the `PBORCA_SessionOpen` function. Most of the function calls are fairly straightforward. Others, like those requiring callbacks, are a bit more complicated.

For information about callback functions, see [About ORCA callback functions](#).

1.6.1.4 Final step: close the session

The last step in an ORCA program is to close the session. This allows the Library Manager to clean up and free all resources associated with the session.

This sample C function closes the session:

```
void WINAPI SessionClose(hORCASession)
{
    PBORCA_SessionClose(hORCASession);
    return;
}
```

1.6.2 Bootstrapping a new application

Beginning with PowerBuilder 5.0, you can use ORCA to create the libraries for an entire application from object source code. You don't need to start with an existing PBL.

To import an object, ordinarily you need a library with an Application object that already exists. When you set the Application object to a NULL value during the bootstrap process, ORCA uses a temporary Application object so that you can import your own Application object. But your Application object doesn't become the current application until you close the session, start a new session, and set the current application.

To bootstrap a new application:

1. Start an ORCA session using `PBORCA_SessionOpen`.
2. Create the new library using `PBORCA_LibraryCreate`.
3. Set the library list for the session to the new library using `PBORCA_SessionSetLibraryList`.
4. Pass NULL variables as the library name and application name with `PBORCA_SessionSetCurrentAppl`.
5. Import the Application object into the new library using `PBORCA_CompileEntryImportList`.

Do not import other objects now

Why you should import only the Application object

Although you can import additional objects into the library, it is not a good idea. In the bootstrap session, the default Application object is the current application. If the objects have any dependencies on your Application object (for example, if they reference global variables), they will cause errors and fail to be imported.

6. Close the session.

Finishing the bootstrapped application

The bootstrap process gets you started with the new application. To complete the process, you need to import the rest of the objects into one or more libraries.

You can only set the library list and current application once in a session, so you need to start a new ORCA session to finish the process. Since you now have a library with the Application object you want to use, the process is the same as any other ORCA session that imports objects.

To finish the bootstrapped application:

1. Open another ORCA session.
 2. Create any additional libraries you'll need for the application.
 3. Set the library list to the library created in the bootstrap procedure plus the empty libraries just created.
 4. Set the current application to the Application object imported in the bootstrap procedure.
 5. Import objects into each of the libraries as needed.
-

When to create the libraries

You can create the additional libraries during the first bootstrap procedure. However, you should not import objects until the second procedure, when the correct Application object is current.

1.7 Removing obsolete ORCA functions

PowerBuilder 8 introduced a new way of accessing source control using the SCC API. The ORCA functions for working with source control were obsolete, but were not removed from the ORCA 8 API.

Starting with PowerBuilder 9, new ORCA source control functions have been added and old ORCA source control functions have been removed from the ORCA API. Therefore, you must remove all calls to the following functions from your existing ORCA applications:

- `PBORCA_CheckOutEntry`

- PBORCA_CheckInEntry
- PBORCA_ListCheckOutEntries

New ORCA functions are documented in [ORCA Functions](#).

2 ORCA Functions

About this chapter

This chapter documents the ORCA functions.

2.1 About the examples

The examples in this chapter assume that a structure was set up to store information about the ORCA session when the session was opened. In the examples, the variable `lpORCA_Info` is a pointer to an instance of this structure:

```
typedef struct ORCA_Info {
    LPTSTR lpszErrorMessage; // Ptr to message text
    HPBORCA hORCASession; // ORCA session handle
    DWORD dwErrorBufferLen; // Length of error buffer
    long lReturnCode; // Return code
    HINSTANCE hLibrary; // Handle to ORCA library
    PPBORCA_CONFIG_SESSION pConfig; // ConfigureSession
} ORCA_INFO, FAR *PORCA_INFO;
```

2.2 ORCA return codes

The header file `PBORCA.H` defines these return codes:

Table 2.1:

Return code	Description
0 <code>PBORCA_OK</code>	Operation successful
-1 <code>PBORCA_INVALIDPARMS</code>	Invalid parameter list
-2 <code>PBORCA_DUPOPERATION</code>	Duplicate operation
-3 <code>PBORCA_OBJNOTFOUND</code>	Object not found
-4 <code>PBORCA_BADLIBRARY</code>	Bad library name
-5 <code>PBORCA_LIBLISTNOTSET</code>	Library list not set
-6 <code>PBORCA_LIBNOTINLIST</code>	Library not in library list
-7 <code>PBORCA_LIBIOERROR</code>	Library I/O error
-8 <code>PBORCA_OBJEXISTS</code>	Object exists
-9 <code>PBORCA_INVALIDNAME</code>	Invalid name
-10 <code>PBORCA_BUFFERTOOSMALL</code>	Buffer size is too small
-11 <code>PBORCA_COMPERROR</code>	Compile error
-12 <code>PBORCA_LINKERROR</code>	Link error
-13 <code>PBORCA_CURRAPPLNOTSET</code>	Current application not set
-14 <code>PBORCA_OBJHASNOANCS</code>	Object has no ancestors
-15 <code>PBORCA_OBJHASNOREFS</code>	Object has no references
-16 <code>PBORCA_PBD COUNTERROR</code>	Invalid # of PBDs
-17 <code>PBORCA_PBD CREATERROR</code>	PBD create error

Return code	Description
-18 PBORCA_CHECKOUTERROR	Source Management error (obsolete)
-19 PBORCA_CBCREATEERROR	Could not instantiate ComponentBuilder class
-20 PBORCA_CBINITERROR	Component builder Init method failed
-21 PBORCA_CBBUILDERROR	Component builder BuildProject method failed
-22 PBORCA_SCCFAILURE	Could not connect to source control
-23 PBORCA_REGREADERROR	Could not read registry
-24 PBORCA_SCCLOADDLLFAILED	Could not load DLL
-25 PBORCA_SCCINITFAILED	Could not initialize SCC connection
-26 PBORCA_OPENPROJFAILED	Could not open SCC project
-27 PBORCA_TARGETNOTFOUND	Target File not found
-28 PBORCA_TARGETREADERR	Unable to read Target File
-29 PBORCA_GETINTERFACEERROR	Unable to access SCC interface
-30 PBORCA_IMPORTONLY_REQ	Scs connect offline requires IMPORTONLY refresh option
-31 PBORCA_GETCONNECT_REQ	SCC connect offline requires GetConnectProperties with Exclude_Checkout
-32 PBORCA_PBCFILE_REQ	SCC connect offline with Exclude_Checkout requires PBC file

2.3 PBORCA_ApplicationRebuild

Description

Compiles all the objects in the libraries included on the library list. If necessary, the compilation is done in multiple passes to resolve circular dependencies.

Syntax

```
INT PBORCA_ApplicationRebuild ( HPBORCA hORCASession,
    PBORCA_REBLD_TYPE eRebldType,
    PBORCA_ERRPROC pCompErrProc,
    LPVOID pUserData );
```

Table 2.2:

Argument	Description
hORCASession	Handle to previously established ORCA session.
eRebldType	A value of the PBORCA_REBLD_TYPE enumerated data type specifying the type of rebuild. Values are: PBORCA_FULL_REBUILD

Argument	Description
	PBORCA_INCREMENTAL_REBUILD PBORCA_MIGRATE PBORCA_3PASS
pCompErrorProc	<p>Pointer to the PBORCA_ApplicationRebuild callback function. The callback function is called for each error that occurs as the objects are compiled.</p> <p>The information ORCA passes to the callback function is error level, message number, message text, line number, and column number, stored in a structure of type PBORCA_COMPERR. The object name and script name are part of the message text.</p> <p>If you do not want to use a callback function, set pCompErrorProc to 0.</p>
pUserData	<p>Pointer to user data to be passed to the PBORCA_CompileEntryImport callback function.</p> <p>The user data typically includes the buffer or a pointer to the buffer in which the callback function stores the error information as well as information about the size of the buffer.</p> <p>If you are not using a callback function, set pUserData to 0.</p>

Return value

INT. Typical return codes are:

Table 2.3:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-13 PBORCA_CURRAPPLNOTSET	Current application not set

Usage

You must set the library list and current application before calling this function.

If you use the compile functions, errors can occur because of the order in which the objects are compiled. If two objects refer to each other, then simple compilation will fail. Use PBORCA_ApplicationRebuild to resolve errors due to object dependencies.

PBORCA_ApplicationRebuild resolves circular dependencies with multiple passes through the compilation process.

The rebuild types specify how objects are affected. Choices are:

Incremental rebuild

Updates all the objects and libraries referenced by any objects that have been changed since the last time you built the application.

Full rebuild

Updates all the objects and libraries in your application.

Migrate

Updates all the objects and libraries in your application to the current version. Only applicable when the objects were built in an earlier version.

Examples

This example recompiles all the objects in the libraries on the current library list.

Each time an error occurs, PBORCA_ApplicationRebuild calls the callback CompileEntryErrors. In the code you write for CompileEntryErrors, you store the error messages in the buffer pointed to by lpUserData:

```
PBORCA_ERRPROC fpError;  
int nReturnCode;  
  
fpError = (PBORCA_ERRPROC) ErrorProc;  
nReturnCode = PBORCA_ApplicationRebuild(  
    lpORCA_Info->hORCASession,  
    PBORCA_FULL_REBUILD,  
    fpError, lpUserData);
```

For more information about setting up the data buffer for the callback, see [Content of a callback function](#) and the example for [PBORCA_LibraryDirectory](#).

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA_CompileEntryRegenerate](#)

[PBORCA_CompileEntryImport](#)

[PBORCA_CompileEntryImportList](#)

2.4 PBORCA_BuildProject

Description

This function is obsolete because EAServer is no longer supported since PowerBuilder 2017.

Deploys an EAServer component according to the specifications of the project object.

Syntax

```
INT PBORCA_BuildProject ( HPBORCA hORCASession,
```

```
LPTSTR lpszLibraryName,
LPTSTR lpszProjectName,
PBORCA_BLDPROC pBuildErrProc,
LPVOID pUserData );
```

Table 2.4:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	File name of the library containing project entry.
lpszProjectName	Project object containing deployment information.
pBuildErrProc	Pointer to the PBORCA_BuildProject error callback function. If you don't want to use a callback function, set pBuildErrProc to NULL.
pUserData	Pointer to user data to be passed to the callback function.

Return value

INT. Typical return codes are:

Table 2.5:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-19 PBORCA_CBCREATEERROR	Component Builder class not created
-20 PBORCA_CBINITERROR	Initialization of EA Server connection failed
-21 PBORCA_CBBUILDERROR	Deployment failed with errors

Usage

How error information is returned

PBORCA_BuildProject error callback function stores information about an entry in the following structure. You pass a pointer to the structure in the pBuildErrProc argument:

```
typedef struct PBORCA_blderr
{
    LPTSTR lpszMessageText; // Pointer to message text
} PBORCA_BLDERR, FAR *PPBORCA_BLDERR;
```

Prototype for callback function

The callback function has the following signature:

```
typedef PBCALLBACK (void, *PPBORCA_BLDPROC) (PBORCA_BLDERR, LPVOID);
```

See also

[PBORCA_BuildProjectEx](#)

2.5 PBORCA_BuildProjectEx

Description

This function is obsolete because EAServer is no longer supported since PowerBuilder 2017.

Deploys an EAServer component according to the specifications of the project object, but overrides server and port properties in the project object with the argument values you specify. However, it does not override these properties if they are set in the server profile. To override properties in the server profile and the project object, use `PBORCA_BuildProjectWithOverrides`.

Syntax

```
INT PBORCA_BuildProjectEx ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszProjectName,
    PBORCA_BLDPROC pBuildErrProc,
    LPTSTR lpszServerName,
    INT iPort,
    LPVOID pUserData );
```

Table 2.6:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	File name of the library containing project entry.
lpszProjectName	Project object containing deployment information.
pBuildErrProc	Pointer to the <code>PBORCA_BuildProject</code> error callback function. If you don't want to use a callback function, set <code>pBuildErrProc</code> to <code>NULL</code> .
lpszServerName	Server name for EAServer deployment. This value overrides the server property in the project object.
iPort	Port number for EAServer deployment. This value overrides the server property in the project object.
pUserData	Pointer to user data to be passed to the callback function.

Return value

INT. Typical return codes are:

Table 2.7:

Return code	Description
0 PBORCA_OK	Operation successful.
-1 PBORCA_INVALIDPARMS	Invalid parameter list.
-19 PBORCA_CBCREATEERROR	Component Builder class not created.
-20 PBORCA_CBINITERROR	Initialization of EAServer connection failed.
-21 PBORCA_CBBUILDERROR	Deployment failed with errors.

See also

[PBORCA_BuildProject](#)

[PBORCA_BuildProjectWithOverrides](#)

2.6 PBORCA_BuildProjectWithOverrides

Description

This function is obsolete because EAServer is no longer supported since PowerBuilder 2017.

Deploys an EAServer component according to the specifications of the project object, but forces overrides based on argument values you specify. This method is similar to PBORCA_BuildProjectEx, however, it requires additional input values for the server login ID and password, and it uses these values to override any values set in the server profile or the project object.

Syntax

```
INT PBORCA_BuildProjectWithOverrides ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszProjectName,
    PBORCA_BLDPROC pBuildErrProc,
    LPTSTR lpszServerName,
    INT iPort,
    LPTSTR lpszUserid,
    LPTSTR lpszPassword,
    LPVOID pUserData );
```

Table 2.8:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	File name of the library containing project entry.
lpszProjectName	Project object containing deployment information.

Argument	Description
pBuildErrProc	Pointer to the PBORCA_BuildProject error callback function. If you don't want to use a callback function, set pBuildErrProc to NULL.
lpszServerName	Server name for EAServer deployment. This value overrides the server property in the project object.
iPort	Port number for EAServer deployment. This value overrides the server property in the project object.
lpszUserid	Login ID for the server. This value overrides the login ID in the project object.
lpszPassword	Password for the server. This value overrides the login password in the project object.
pUserData	Pointer to user data to be passed to the callback function.

Return value

INT. Typical return codes are:

Table 2.9:

Return code	Description
0 PBORCA_OK	Operation successful.
-1 PBORCA_INVALIDPARMS	Invalid parameter list.
-19 PBORCA_CBCREATEERROR	Component Builder class not created.
-20 PBORCA_CBINITERROR	Initialization of EAServer connection failed.
-21 PBORCA_CBBUILDERROR	Deployment failed with errors.

See also

[PBORCA_BuildProject](#)

[PBORCA_BuildProjectEx](#)

2.7 PBORCA_CompileEntryImport

Description

Imports the source code for a PowerBuilder object into a library and compiles it.

Syntax

```
INT PBORCA_CompileEntryImport ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszEntryName,
    PBORCA_TYPE otEntryType,
```

```

    lpszComments,
    LPTSTR lpszEntrySyntax,
    LONG lEntrySyntaxBuffSize,
    PBORCA_ERRPROC pCompErrorProc,
    LPVOID pUserData );

```

Table 2.10:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	Pointer to a string whose value is the file name of the library into which you want to import the object.
lpszEntryName	Pointer to a string whose value is the name of the object being imported.
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being imported. Values are: PBORCA_APPLICATION PBORCA_BINARY PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW
lpszComments	Pointer to a string whose value is the comments you are providing for the object.
lpszEntrySyntax	Pointer to a buffer whose value is source code for the object to be imported. If an export header exists in the source code it is ignored. The source encoding for lpszEntrySyntax is specified by the eImportEncoding property in the PBORCA_CONFIG_SESSION structure.
lEntrySyntaxBuffSize	Length of the lpszEntrySyntax buffer. This length is specified in bytes regardless of the source encoding.

Argument	Description
pCompErrorProc	<p>Pointer to the PBORCA_CompileEntryImport callback function. The callback function is called for each error that occurs as the imported object is compiled.</p> <p>The information ORCA passes to the callback function is error level, message number, message text, line number, and column number, stored in a structure of type PBORCA_COMPERR. The object name and script name are part of the message text.</p> <p>If you don't want to use a callback function, set pCompErrorProc to 0.</p>
pUserData	<p>Pointer to user data to be passed to the PBORCA_CompileEntryImport callback function.</p> <p>The user data typically includes the buffer or a pointer to the buffer in which the callback function stores the error information as well as information about the size of the buffer.</p> <p>If you are not using a callback function, set pUserData to 0.</p>

Return value

INT. Typical return codes are:

Table 2.11:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-4 PBORCA_BADLIBRARY	Bad library name, library not found, or object could not be saved in the library
-6 PBORCA_LIBNOTINLIST	Library not in list
-8 PBORCA_COMPERROR	Compile error
-9 PBORCA_INVALIDNAME	Name does not follow PowerBuilder naming rules
-13 PBORCA_CURRAPPLNOTSET	The current application has not been set

Usage

You must set the library list and current Application object before calling this function.

PowerBuilder

In PowerBuilder 10 and higher, you must specify the source encoding for the objects to be imported. You do this by setting the `eImportEncoding` property in the `PBORCA_CONFIG_SESSION` structure and calling `PBORCA_ConfigureSession`. For ANSI clients the default source encoding is ANSI/DBCS; for Unicode clients the default source encoding is Unicode.

Importing objects with embedded binary information

Two separate calls to `PBORCA_CompileEntryImport` are required to import objects containing embedded binary data such as OLE objects. The first call imports the source component. The second call imports the binary component using an `otEntryType` argument set to `PBORCA_BINARY` and an `lpzEntrySyntax` argument pointing to the start of the binary header record.

When errors occur

When errors occur during importing, the object is brought into the library but might need editing. An object with minor errors can be opened in its painter for editing. If the errors are severe enough, the object can fail to open in the painter and you will have to export the object, fix the source code, and import it again. If errors are due to the order in which the objects are compiled, you can call the `PBORCA_ApplicationRebuild` function after all the objects are imported.

Caution

When you import an entry with the same name as an existing entry, the old entry is deleted before the import takes place. If an import fails, the old object will already be deleted.

For information about callback processing for errors, see [PBORCA_CompileEntryImportList](#).

Examples

This example imports a DataWindow called `d_labels` into the library `DWOBJECTS.PBL`. The source code is stored in a buffer called `szEntrySource`.

Each time an error occurs, `PBORCA_CompileEntryImport` calls the callback `CompileEntryErrors`. In the code you write for `CompileEntryErrors`, you store the error messages in the buffer pointed to by `lpUserData`:

```
PBORCA_ERRPROC fpError;
int nReturnCode;

fpError = (PBORCA_ERRPROC) ErrorProc;
nReturnCode = PBORCA_CompileEntryImport(
    lpORCA_Info->hORCASession,
    _TEXT("c:\\app\\dwobjects.pbl"),
    _TEXT("d_labels"), PBORCA_DATAWINDOW,
    (LPTSTR) szEntrySource, 60000,
    fpError, lpUserData);
```

In these examples, session information is saved in the data structure `ORCA_Info`, shown in [About the examples](#).

This example reads a source file, determines the encoding format of the source file, and imports it into a PBL. If the file contains an embedded binary object, this is also imported using a second call to `PBORCA_CompileEntryImport`.

```
// Headers, Defines, Typdefs
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <tchar.h>
extern "C" {
#include "pborca.h"
}
// Global Variables
HPBORCA hPbOrca;
PBORCA_ERRPROC fpError;
// Function Declarations
void CALLBACK ErrorProc(PBORCA_COMPERR *lpCompErr,
LPVOID lpUserData);

// NAME: Impbin.cpp
// Synopsis: Import w_edit_connect.srw (which contains
// an embedded OLE object) into a work PBL.
// This example can be compiled as an ANSI client
// or a Unicode client. To compile as Unicode
// use /DUNICODE /D_UNICODE compiler directives.
#ifdef UNICODE
INT wmain ( int argc, wchar_t *argv[])
#else
INT main ( int argc, char *argv[])
#endif
{
LPTSTR pszLibraryName[5];
LPTSTR pszImportFile;
HANDLE hOpenFile = NULL;
INT iErrCode;
BOOL rc;
wchar_t chMarker;
unsignedchar chMarker3;
DWORD dBytesRead;
DWORD dFileSize;
PBORCA_CONFIG_SESSION Config;
LPBYTE pReadBuffer = NULL;
LPBYTE pEndBuffer;
INT iSourceSize;
INT iBinarySize;
pszLibraryName[0] = _TEXT("c:\\pb12.5\\main\\pbls\\qadb\\qadbtest\\qadbtest.pbl");
pszLibraryName[1] = _TEXT("c:\\pb12.5\\main\\pbls\\qadb\\shared_obj\\
\\shared_obj.pbl");
pszLibraryName[2] = _TEXT("c:\\pb12.5\\main\\pbls\\qadb\\datatypes\\datatype.pbl");
pszLibraryName[3] = _TEXT("c:\\pb12.5\\main\\pbls\\qadb\\chgreqs\\chgreqs.pbl");
pszLibraryName[4] = _TEXT("c:\\pb12.5\\main\\orca\\testexport\\work.pbl");
pszImportFile = _TEXT("c:\\pb12.5\\main\\pbls\\qadb\\qadbtest\\
\\w_edit_connect.srw");
memset(&Config, 0x00, sizeof(PBORCA_CONFIG_SESSION));
PbOrca = PBORCA_SessionOpen();
// Delete and re-create work.pbl
iErrCode = PBORCA_LibraryDelete(hPbOrca, pszLibraryName[4]);
iErrCode = PBORCA_LibraryCreate(hPbOrca,
pszLibraryName[4], _TEXT("work pbl"));
iErrCode = PBORCA_SessionSetLibraryList(hPbOrca,
pszLibraryName, 5);

if (iErrCode != PBORCA_OK)
```

```

goto TestExit;
iErrCode = PBORCA_SessionSetCurrentAppl(hPbOrca,
    pszLibraryName[0], _TEXT("qadbtest"));
if (iErrCode != PBORCA_OK)
    goto TestExit;
// PBORCA_CompileEntryImport ignores export headers,
// so the ORCA application must programmatically
// determine the source encoding of the import file.
// This is done by reading the first two or three
// bytes of the file.
hOpenFile = CreateFile(pszImportFile, GENERIC_READ, 0,
    NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if( hOpenFile == INVALID_HANDLE_VALUE )
    goto TestExit;
rc = ReadFile(hOpenFile, (LPVOID)&chMarker,
    sizeof(wchar_t), &dBytesRead, NULL);
if( rc )
{
    if (chMarker == 0xfeff)
        Config.eImportEncoding = PBORCA_UNICODE;
    else if (chMarker == 0xbef)
    {
        rc = ReadFile(hOpenFile, (LPVOID)&chMarker3,
            sizeof(CHAR), &dBytesRead, NULL);
        if (chMarker3 == 0xbf)
            Config.eImportEncoding = PBORCA_UTF8;
    }
    else if (memcmp((LPBYTE) &chMarker, "HA", 2) == 0)
        Config.eImportEncoding = PBORCA_HEXASCII;
    else
        Config.eImportEncoding = PBORCA_ANSI_DBCS;

// Now allocate memory for a source buffer and read
// entire file
SetFilePointer( hOpenFile, 0, NULL, FILE_BEGIN);
dFileSize = GetFileSize(hOpenFile, NULL) ;
pReadBuffer = (LPBYTE) malloc((size_t) dFileSize + 2);
rc = ReadFile(hOpenFile, pReadBuffer, dFileSize,
    &dBytesRead, NULL);
// Append a null terminator to enable strstr() call
pEndBuffer = pReadBuffer + dFileSize;
memset(pEndBuffer, 0x00, 2); // unicode EOF marker
if (!rc)
    goto TestExit;
// Determine if the object includes a binary component.
// If it does, then make two separate calls to
// PBORCA_CompileEntryImport.
if (Config.eImportEncoding == PBORCA_UNICODE)
{
    LPWSTR
        pszUniBinHeader;
    LPWSTR
        pUniBinStart;
    pszUniBinHeader = "Start of PowerBuilder Binary
        Data Section";
    pUniBinStart = wcsstr((const wchar_t *)
        pReadBuffer, pszUniBinHeader);

    if (pUniBinStart)
    {
        pEndBuffer = (LPBYTE) pUniBinStart;
        iSourceSize = (INT) (pEndBuffer - pReadBuffer);
        iBinarySize = (INT) (dFileSize - iSourceSize);
    }
}
}

```

```

    else
    {
        iSourceSize = (INT) dFileSize;
        iBinarySize = 0;
    }
}
else
{
    LPSTR pszAnsiBinHeader;
    LPSTR pAnsiBinStart;
    pszAnsiBinHeader = "Start of PowerBuilder Binary
        Data Section";
    pAnsiBinStart = (LPSTR) strstr((const char *)
        pReadBuffer, (const char *) pszAnsiBinHeader);
    if (pAnsiBinStart)
    {
        pEndBuffer = (LPBYTE) pAnsiBinStart;
        iSourceSize = (INT) (pEndBuffer - pReadBuffer);
        iBinarySize = (INT) (dFileSize - iSourceSize);
    }
    else
    {
        iSourceSize = (INT) dFileSize;
        iBinarySize = 0;
    }
}
// Configure ORCA session to read appropriate source
// encoding
iErrCode = PBORCA_ConfigureSession(hPbOrca, &Config);

// Now import the source for the entry
fpError = (PBORCA_ERRPROC) ErrorProc;
iErrCode = PBORCA_CompiledEntryImport(
    hPbOrca,
    pszLibraryName[4],
    _TEXT("w_edit_connect"), PBORCA_WINDOW,
    _TEXT("test embedded OLE object"),
    (LPTSTR) pReadBuffer, iSourceSize,
    fpError, NULL);
if (iErrCode != PBORCA_OK)
    goto TestExit;
if (iBinarySize > 0)
{
    iErrCode = PBORCA_CompiledEntryImport(
        hPbOrca,
        pszLibraryName[4],
        _TEXT("w_edit_connect"), PBORCA_BINARY,
        NULL,
        (LPTSTR) pEndBuffer, iBinarySize,
        fpError, NULL);
}
}
TestExit:
if ( hOpenFile != INVALID_HANDLE_VALUE )
    CloseHandle(hOpenFile);
if (pReadBuffer)
    free(pReadBuffer);
PBORCA_SessionClose(hPbOrca);
return iErrCode;
}
// Callback error procedure used by the call to compile
// an object. In this example it is supplied by the
// program and is not a method of the ORCA class.
void CALLBACK ErrorProc(PBORCA_COMPERR *lpCompErr,

```

```

LPVOID lpUserData)
{
_tprintf(_TEXT("%s \n"), lpCompErr->lpszMessageText );
}

```

See also[PBORCA LibraryEntryExport](#)[PBORCA CompileEntryImportList](#)[PBORCA CompileEntryRegenerate](#)[PBORCA ApplicationRebuild](#)

2.8 PBORCA_CompileEntryImportList

Description

Imports the source code for a list of PowerBuilder objects into libraries and compiles them. The name of each object to be imported is held in an array. Other arrays hold the destination library, object type, comments, and source code. The arrays must have an element for every object.

Syntax

```

INT PBORCA_CompileEntryImportList ( PBORCA hORCASession,
LPTSTR far *pLibraryNames,
LPTSTR far *pEntryNames,
PBORCA_TYPE far *otEntryTypes,
LPTSTR far *pComments,
LPTSTR far *pEntrySyntaxBuffers,
LONG far *pEntrySyntaxBuffSizes,
INT iNumberOfEntries,
PBORCA_ERRPROC pCompErrorProc,
LPVOID pUserData );

```

Table 2.12:

Argument	Description
hORCASession	Handle to previously established ORCA session.
*pLibraryNames	Pointer to an array of strings whose values are the file names of libraries into which you want to import the corresponding objects.
*pEntryNames	Pointer to an array of strings whose values are the names of objects to be imported into the corresponding libraries.
*otEntryTypes	Pointer to an array whose values are the object types of the library entries, expressed as enumerated data type PBORCA_TYPE. Values are: PBORCA_APPLICATION PBORCA_DATAWINDOW PBORCA_FUNCTION

Argument	Description
	PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT PBORCA_BINARY
*pComments	Pointer to an array of strings whose values are the comments for the corresponding objects.
*pEntrySyntaxBuffers	Pointer to an array of strings whose values are the source code for the corresponding objects.
*pEntrySyntaxBuffSizes	Pointer to an array of longs whose values are the lengths of the strings pointed to by *pEntrySyntaxBuffers
iNumberOfEntries	Number of entries to be imported, which is the same as the array length of all the array arguments.
pCompErrorProc	Pointer to the PBORCA_CompiledEntryImportList callback function. The callback function is called for each error that occurs when imported objects are compiled. The information ORCA passes to the callback function is error level, message number, message text, line number, and column number, stored in a structure of type PBORCA_COMPERR. The object name and script name are part of the message text. If you don't want to use a callback function, set pCompErrorProc to 0.
pUserData	Pointer to user data to be passed to the PBORCA_CompiledEntryImportList callback function. The user data typically includes the buffer or a pointer to the buffer in which the callback

Argument	Description
	function formats the error information as well as information about the size of the buffer. If you are not using a callback function, set pUserData to 0.

Return value

INT. Typical return codes are:

Table 2.13:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-4 PBORCA_BADLIBRARY	Bad library name, library not found, or object couldn't be saved in the library
-6 PBORCA_LIBNOTINLIST	Library not in list
-7 PBORCA_LIBIOERROR	Library I/O error
-8 PBORCA_COMPERROR	Compile error
-9 PBORCA_INVALIDNAME	Name does not follow PowerBuilder naming rules
-13 PBORCA_CURRAPPLNOTSET	The current application has not been set

Usage

You must set the library list and current Application object before calling this function.

PBORCA_CompileEntryImportList is useful for importing several interrelated objects -- for example, a window, its menu, and perhaps a user object that it uses.

How imported objects are processed

ORCA imports all the objects in the list, compiling each object type definition. If no errors occur, then ORCA compiles all the objects in all the listed libraries.

Object dependencies

In the list of objects to be imported, put ancestor objects before their descendant objects so that the ancestors are imported first.

In the list of objects, put a user object before objects that refer to that user object so that the referenced object is imported first.

If objects refer to each other, call PBORCA_ApplicationRebuild to get an error-free compilation.

Populating the information arrays for imported objects

The information for each imported object is contained in several parallel arrays. For example, if a DataWindow named `d_labels` is the third element in the object name array (subscript 2), then a pointer to the name of its destination library is the third element in the library name array; its object type is the third element in the object type array; and the pointer to its source code buffer is the third element in the syntax buffer array.

Using `PBORCA_BINARY` to specify entry type

This value of the `PBORCA_TYPE` enumerated data type should be used when importing or exporting entries that contain embedded binary information such as OLE objects. The binary information is imported from a buffer previously filled on export with the hexascii representation of the binary data.

For sample code demonstrating using `PBORCA_BINARY` on import, see [Examples \[35\]](#).

When errors occur

When errors occur during importing, the object is brought into the library but may need editing. An object with minor errors can be opened in its painter for editing. If the errors are severe enough, the object can fail to open in the painter, and you will have to export the object, fix the source code, and import it again. If errors are due to the order in which the objects are compiled, you can call the `PBORCA_ApplicationRebuild` function after all the objects are imported.

Caution

When you import an entry with the same name as an existing entry, the old entry is deleted before the import takes place. If an import fails, the old object will already have been deleted.

Processing errors in the callback function

For each error that occurs during compiling, ORCA calls the callback function pointed to in `pCompErrorProc`. How that error information is returned to your calling program depends on the processing you provide in the callback function. ORCA passes information to the callback function about an error in the structure `PBORCA_COMPERR`. The callback function can examine that structure and store any information it wants in the buffer pointed to by `pUserData`.

Because you do not know how many errors will occur, it is hard to predict the size of the `pUserData` buffer. It is up to your callback function to keep track of the available space in the buffer.

Examples

This example builds the arrays required to import three objects into two libraries (the example assumes that source code for the objects has already been set up in the variables `szWindow1`, `szWindow2`, and `szMenu1`) and imports the objects.

Each time an error occurs, `PBORCA_CompiledEntryImportList` calls the callback `CompileEntryErrors`. In the code you write for `CompileEntryErrors`, you store the error messages in the buffer pointed to by `lpUserData`. In the example, the `lpUserData` buffer has already been set up:

```
LPTSTR lpLibraryNames[3];
```

```

LPTSTR lpObjectNames[3];
PBORCA_TYPE ObjectTypes[3];
LPTSTR lpObjComments[3];
LPTSTR lpSourceBuffers[3];
long BuffSizes[3];
PBORCA_ERRPROC fpError;
int nReturnCode;

fpError = (PBORCA_ERRPROC) ErrorProc;
// Indicate Unicode source encoding
lpORCA_Info->pConfig->eImportEncoding = PBORCA_UNICODE;
PBORCA_ConfigureSession(lpORCA_Info->hORCASession,
    lpORCA_Info->pConfig);

// specify the library names
lpLibraryNames[0] =
    _TEXT("c:\\apeon\\pb2019\\demo\\windows.pbl");
lpLibraryNames[1] =
    _TEXT("c:\\apeon\\pb2019\\demo\\windows.pbl");
lpLibraryNames[2] =
    _TEXT("c:\\apeon\\pb2019\\demo\\menus.pbl");

// specify the object names
lpObjectNames[0] = _TEXT("w_ancestor");
lpObjectNames[1] = _TEXT("w_descendant");
lpObjectNames[2] = _TEXT("m_actionmenu");

// set up object type array
ObjectTypes[0] = PBORCA_WINDOW;
ObjectTypes[1] = PBORCA_WINDOW;
ObjectTypes[2] = PBORCA_MENU;

// specify object comments
lpObjComments[0] = _TEXT("Ancestor window");
lpObjComments[1] = _TEXT("descendant window");
lpObjComments[2] = _TEXT("Action menu");

// set pointers to source code
lpSourceBuffers[0] = (LPTSTR) szWindow1;
lpSourceBuffers[1] = (LPTSTR) szWindow2;
lpSourceBuffers[2] = (LPTSTR) szMenu1;

// Set up source code lengths array
BuffSizes[0] = _tcslen(szWindow1)*2;
    //Unicode source buffer
BuffSizes[1] = _tcslen(szWindow2)*2;
    //Size is always in bytes
BuffSizes[2] = _tcslen(szMenu1)*2;

nReturnCode = PBORCA_CompileEntryImportList(
    lpORCA_Info->hORCASession,
    lpLibraryNames, lpObjectNames, ObjectTypes,
    lpObjComments, lpSourceBuffers, BuffSizes, 3,
    fpError, lpUserData );

```

For more information about setting up the data buffer for the callback, see [Content of a callback function](#) and the example for [PBORCA_LibraryDirectory](#).

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA_LibraryEntryExport](#)

[PBORCA CompileEntryImport](#)

[PBORCA CompileEntryRegenerate](#)

[PBORCA ApplicationRebuild](#)

2.9 PBORCA_CompileEntryRegenerate

Description

Compiles an object in a PowerBuilder library.

Syntax

```
INT PBORCA_CompileEntryRegenerate ( PBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszEntryName,
    PBORCA_TYPE otEntryType,
    PBORCA_ERRPROC pCompErrorProc,
    LPVOID pUserData );
```

Table 2.14:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	Pointer to a string whose value is the file name of the library containing the object to be compiled.
lpszEntryName	Pointer to a string whose value is the name of the object to be compiled.
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being compiled. Values are: PBORCA_APPLICATION PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT
pCompErrorProc	Pointer to the PBORCA_CompileEntryRegenerate callback

Argument	Description
	<p>function. The callback function is called for each error that occurs as the object is compiled.</p> <p>The information ORCA passes to the callback function is error level, message number, message text, line number, and column number, stored in a structure of type <code>PBORCA_COMPERR</code>. The object name and script name are part of the message text.</p> <p>If you don't want to use a callback function, set <code>pCompErrorProc</code> to 0.</p>
pUserData	<p>Pointer to user data to be passed to the <code>PBORCA_CompiledEntryRegenerate</code> callback function.</p> <p>The user data typically includes the buffer or a pointer to the buffer in which the callback function stores the error information as well as information about the size of the buffer.</p> <p>If you are not using a callback function, set <code>pUserData</code> to 0.</p>

Return value

INT. Typical return codes are:

Table 2.15:

Return code	Description
0 <code>PBORCA_OK</code>	Operation successful
-1 <code>PBORCA_INVALIDPARMS</code>	Invalid parameter list
-3 <code>PBORCA_OBJNOTFOUND</code>	Object not found
-4 <code>PBORCA_BADLIBRARY</code>	Bad library name
-5 <code>PBORCA_LIBLISTNOTSET</code>	Library list not set
-6 <code>PBORCA_LIBNOTINLIST</code>	Library not in library list
-7 <code>PBORCA_LIBIOERROR</code>	Library I/O error
-11 <code>PBORCA_COMPERROR</code>	Compile error

Usage

You must set the library list and current Application object before calling this function.

When errors occur

In order to fix errors that occur during the regeneration, you need to export the source code, fix the errors, and import the object, repeating the process until it compiles correctly.

Sometimes you can open objects with minor errors in a PowerBuilder painter and fix them, but an object with major errors must be exported and fixed.

For information about callback processing for errors, see [PBORCA CompileEntryImportList](#).

Examples

This example compiles a DataWindow called d_labels in the library DWOBJECTS.PBL.

Each time an error occurs, PBORCA_CompileEntryRegenerate calls the callback CompileEntryErrors. In the code you write for CompileEntryErrors, you store the error messages in the buffer pointed to by lpUserData. In the example, the lpUserData buffer has already been set up:

```
PBORCA fpError;
int nReturnCode;
fpError = (PBORCA_ERRPROC) ErrorProc;
nReturnCode = PBORCA_CompileEntryRegenerate(
    lpORCA_Info->hORCASession,
    _TEXT("c:\\app\\dwobjects.pbl"),
    _TEXT("d_labels"), PBORCA_DATAWINDOW,
    fpError, lpUserData );
```

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA LibraryEntryExport](#)

[PBORCA CompileEntryImport](#)

[PBORCA CompileEntryImportList](#)

[PBORCA ApplicationRebuild](#)

2.10 PBORCA_ConfigureSession

Description

PBORCA_ConfigureSession facilitates backward compatibility with PowerBuilder 10. It increases the flexibility of the API and minimizes the changes necessary to other ORCA function signatures.

Syntax

```
INT PBORCA_ConfigureSession ( PBORCA hORCASession, PPBORCA_CONFIG_SESSION
    pSessionConfig );
```

Table 2.16:

Argument	Description
hORCASession	Handle to previously established ORCA session.
pSessionConfig	Structure that lets the ORCA client specify the behavior of subsequent requests. Settings remain in effect for the

Argument	Description
	duration of the session or until you call PBORCA_ConfigureSession again. Be sure to specify all of the settings each time you call PBORCA_ConfigureSession.

Return value

INT. Typical return codes are:

Table 2.17:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Session not open or null pConfig pointer

Usage

Create an instance of a PBORCA_CONFIG_SESSION structure and populate it with your configuration settings. Then call PBORCA_ConfigureSession immediately after SessionOpen. You can also call this function anytime thereafter to reset configuration properties.

```
typedef enum pborca_clobber
{
    PBORCA_NOCLOBBER,
    PBORCA_CLOBBER,
    PBORCA_CLOBBER_ALWAYS
    PBORCA_CLOBBER_DECIDED_BY_SYSTEM
} PBORCA_ENUM_FILEWRITE_OPTION;

typedef enum pborca_type
{
    PBORCA_UNICODE,
    PBORCA_UTF8,
    PBORCA_HEXASCII,
    PBORCA_ANSI_DBCS
} PBORCA_ENCODING;

typedef struct pborca_configsession
{
    PBORCA_ENUM_FILEWRITE_OPTION
    eClobber; // overwrite existing file?
    PBORCA_ENCODING eExportEncoding;
    // Encoding of exported source
    BOOL bExportHeaders;
    // Format source with export header
    BOOL bExportIncludeBinary; // Include the binary
    BOOL bExportCreateFile; // Export source to a file
    LPTSTR pExportDirectory;
    // Directory for exported files
    PBORCA_ENCODING eImportEncoding;
    // Encoding of imported source
    BOOL bDebug; // Debug compiler directive
    PVOID filler2; // Reserved for future use
    PVOID filler3;
    PVOID filler4;
}
```

```
} PBORCA_CONFIG_SESSION, FAR *PPBORCA_CONFIG_SESSION;
```

Table 2.18:

Member variable	Description
eClobber	<p>Specifies when to overwrite existing files on the file system. This property is used by:</p> <ul style="list-style-type: none"> PBORCA_LibraryEntryExport PBORCA_LibraryEntryExportEx PBORCA_DynamicLibraryCreate PBORCA_ExecutableCreate PBORCA_LibraryDelete <p>You can set any of the following eClobber values for a configuration session:</p> <ul style="list-style-type: none"> • PBORCA_NOCLOBBER never overwrites an existing file • PBORCA_CLOBBER overwrites existing files that are not write-protected • PBORCA_CLOBBER_ALWAYS overwrites existing files that are write-protected • PBORCA_CLOBBER_DECIDED_BY_SYSTEM causes the functions mentioned above to behave as they did in prior ORCA releases
eExportEncoding	<p>Specifies the source encoding used by PBORCA_LibraryEntryExport:</p> <ul style="list-style-type: none"> • PBORCA_UNICODE default for Unicode ORCA clients • PBORCA_ANSI_DBCS default for ANSI ORCA clients • PBORCA_UTF8 • PBORCA_HEXASCII
bExportHeaders	<p>If you set this variable to TRUE, PBORCA_LibraryEntryExport generates export headers. The default value is FALSE for backward compatibility.</p>

Member variable	Description
bExportIncludeBinary	If you set this variable to TRUE, PBORCA_LibraryEntryExport generates the binary component of an object in addition to the source component. The default value is FALSE for backward compatibility.
bExportCreateFile	If you set this variable to TRUE, PBORCA_LibraryEntryExport exports source to a file. The generated file name is the PowerBuilder object entry name with a .sr? file extension. The default value is FALSE.
pExportDirectory	Directory where you export PowerBuilder objects if bExportCreateFile is TRUE.
eImportEncoding	Source encoding. Subsequent calls to PBORCA_CompileEntryImport and PBORCA_CompileEntryImportList expect the lpszEntrySyntax argument to contain this information.
bDebug	If you set this value to FALSE, the DEBUG conditional compiler directive is turned off. All subsequent methods that invoke the PowerScript compiler will use this setting when evaluating script inside DEBUG conditional compilation blocks. This setting is not used in Windows Forms targets, since PBORCA_DeployWinFormProject uses a setting in the Project object of these targets to determine whether to enable or disable the DEBUG directive.

Examples

This example populates the PBORCA_CONFIG_SESSION structure with configuration settings:

```

INT      ConfigureSession(LPTSTR sEncoding)
{
    INT      iErrCode = -1;
    lpORCA_Info->pConfig = (PPBORCA_CONFIG_SESSION)
        malloc(sizeof(PBORCA_CONFIG_SESSION));
    memset(lpORCA_Info->pConfig, 0,
        sizeof(PBORCA_CONFIG_SESSION));

    if (!_tcscmp(sEncoding, _TEXT("ANSI")))
    {
        lpORCA_Info->pConfig->eExportEncoding = PBORCA_ANSI_DBCS;
        lpORCA_Info->pConfig->eImportEncoding = PBORCA_ANSI_DBCS;
    }
    else if (!_tcscmp(sEncoding, _TEXT("UTF8")))
    {

```

```

lpORCA_Info->pConfig->eExportEncoding = PBORCA_UTF8;
lpORCA_Info->pConfig->eImportEncoding = PBORCA_UTF8;
}
else if (!_tcscmp(sEncoding, _TEXT("HEXASCII")))
{
    lpORCA_Info->pConfig->eExportEncoding = PBORCA_HEXASCII;
    lpORCA_Info->pConfig->eImportEncoding = PBORCA_HEXASCII;
}
else
{
    lpORCA_Info->pConfig->eExportEncoding = PBORCA_UNICODE;
    lpORCA_Info->pConfig->eImportEncoding = PBORCA_UNICODE;
}
lpORCA_Info->pConfig->eClobber = PBORCA_CLOBBER;
lpORCA_Info->pConfig->bExportHeaders = TRUE;
lpORCA_Info->pConfig->bExportIncludeBinary = FALSE;
lpORCA_Info->pConfig->bExportCreateFile = FALSE;
lpORCA_Info->pConfig->pExportDirectory = NULL;
lpORCA_Info->pConfig->bDebug = FALSE;
iErrCode = PBORCA_ConfigureSession(
    lpORCA_Info->hORCASession,
    lpORCA_Info->pConfig);
return iErrCode;
}

```

See also[PBORCA_ApplicationRebuild](#)[PBORCA_CompileEntryImportList](#)[PBORCA_SetDebug](#)

2.11 PBORCA_DeployWinFormProject

Description

Generates and compiles Windows Forms project and deploys the assemblies according to the specifications contained in the project objects.

Syntax

```

INT PBORCA_DeployWinFormProject (
    HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszProjectName,
    LPTSTR lpszIconFileName,
    PBORCA_DOTNETPROC pDotNetProc
    LPVOID pUserData );

```

Table 2.19:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	Pointer to a string whose value is the file name containing the project entry.
lpszProjectName	Project object containing deployment information.
lpszIconFileName	Name of the application icon file.

Argument	Description
pDotNetProc	Pointer to the PBORCA_DOTNETPROC callback function. The callback function is called for each message that is generated. All ORCA_ERROR_MESSAGE messages are returned first, followed by all PBORCA_WARNING_MESSAGE messages, and then, by all PBORCA_UNSUPPORTED_FEATURE messages.
pUserData	Pointer to user data to be passed to the PBORCA_DOTNETPROC callback function.

Return value

INT. The typical return codes are:

Table 2.20:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-4 PBORCA_BADLIBRARY	Load library for necessary DLL failed
-5 PBORCA_LIBLISTNOTSET	SessionSetLibraryList is prerequisite
-13 PBORCA_CURRAPPLNOTSET	SessionSetCurrentAppl is prerequisite
-19 PBORCA_CBCREATEERROR	Component builder create error
-20 PBORCA_CBINITERROR	Component builder initialization error
-21 PBORCA_CBBUILDERROR	Component builder build error

Usage

Error information is returned by first creating a callback function associated with PBORCA_DeployWinFormProject that uses the following function signature:

```
void MyDotNetMessageProc (
    PPBORCA_DOTNET_MESSAGE pMsg,
    LPVOID
    pMyUserData)
```

The pMsg argument is a pointer to the following structure:

```
typedef struct pborca_dotnetmsg {
    PBORCA_DOTNET_MSGTYPE
    eMessageType;
    LPTSTR lpszMessageText;
}
PBORCA_DOTNET_MESSAGE FAR *PPBORCA_DOTNET_MESSAGE;
```

The eMessageType argument uses the following enumeration:

```
typedef enum pborca_dotnet_msgtype
```



```
{
  PBORCA_ERROR_MESSAGE,
  PBORCA_WARNING_MESSAGE,
  PBORCA_UNSUPPORTED_FEATURE
} PBORCA_DOTNET_MSGTYPE;
```

Messages are returned to the caller one at a time in the following order: PBORCA_ERROR_MESSAGE messages, PBORCA_WARNING_MESSAGE messages, and PBORCA_UNSUPPORTED_FEATURE messages.

2.12 PBORCA_DynamicLibraryCreate

Description

Creates a PowerBuilder dynamic library (PBD) or PowerBuilder DLL.

Syntax

```
INT PBORCA_DynamicLibraryCreate (
  HPBORCA hORCASession,
  LPTSTR lpszLibraryName,
  LPTSTR lpszPBRName,
  LONG lFlags,
  LPVOID pbcPara = NULL );
```

Table 2.21:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	Pointer to a string whose value is the file name of the library to be built into a PBD or DLL.
lpszPBRName	Pointer to a string whose value is the name of a PowerBuilder resource file whose objects you want to include in the PBD or DLL. If the application has no resource file, specify 0 for the pointer.
lFlags	A long value that indicates which code generation options to apply when building the library. Setting lFlags to 0 generates a native Pcode executable. For information about setting machine code generation options, see PBORCA_ExecutableCreate
pbcPara	Reserved for internal use. Always set pbcPara to NULL.

Return value

INT. The typical return codes are:

Table 2.22:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-4 PBORCA_BADLIBRARY	Bad library name
-17 PBORCA_PBDCREATERROR	PBD create error

Usage

Before calling this function, you must have previously set the library list and current application.

If you plan to build an executable in which some of the libraries are dynamic libraries, you must build those dynamic libraries before building the executable.

Location and name of file

The resulting PBD or DLL will be created in the same directory using the same file name as the PBL. Only the extension changes. For example, for a library C:\DIR1\DIR2\PROG.PBL:

- The output for Pcode is C:\DIR1\DIR2\PROG.PBD
- The output for machine code is C:\DIR1\DIR2\PROG.DLL

eClobber settings

If the PBD or DLL already exists in the file system, the current setting of the eClobber property in the ORCA configuration block (that you set with a PBORCA_ConfigureSession call) determines whether PBORCA_DynamicLibraryCreate succeeds or fails.

Table 2.23:

Current eClobber setting	PBORCA_DynamicLibraryCreate
PBORCA_NOCLOBBER	Fails when an executable file already exists in the file system, regardless of the file attribute settings
PBORCA_CLOBBER or PBORCA_CLOBBER_DECIDED_BY_SYSTEM	Succeeds when the existing executable file has read-write attributes; fails when the executable file has read-only attributes
PBORCA_CLOBBER_ALWAYS	Succeeds regardless of the file attribute settings of an existing executable file

Examples

This example builds a machine code DLL from the library PROCESS.PBL. It is optimized for speed with trace and error context information:

```
LPTSTR pszLibFile;
LPTSTR pszResourceFile;
long lBuildOptions;
int rtn;
// copy file names
pszLibFile = _TEXT("c:\\app\\process.pbl");
pszResourceFile = _TEXT("c:\\app\\process.pbr");
```

```

lBuildOptions = PBORCA_MACHINE_CODE_NATIVE |
    PBORCA_MACHINE_CODE_OPT_SPEED |
    PBORCA_TRACE_INFO | PBORCA_ERROR_CONTEXT;

// create DLL from library
rtn = PBORCA_DynamicLibraryCreate(
    lpORCA_Info->hORCASession,
    pszLibFile, pszResourceFile, lBuildOptions, NULL );

```

In these examples, session information is saved in the data structure `ORCA_Info`, shown in [About the examples](#).

See also

[PBORCA_ConfigureSession](#)

[PBORCA_ExecutableCreate](#)

2.13 PBORCA_ExecutableCreate

Description

Creates a PowerBuilder executable with Pcode or machine code. For a machine code executable, you can request several debugging and optimization options.

The ORCA library list is used to create the application. You can specify which of the libraries have already been built as PBDs or DLLs and which will be built into the executable file.

Syntax

```

INT PBORCA_ExecutableCreate ( HPBORCA hORCASession,
    LPTSTR lpszExeName,
    LPTSTR lpszIconName,
    LPTSTR lpszPBRName,
    PBORCA_LNKPROC pLinkErrProc,
    LPVOID pUserData,
    INT FAR *iPBDFlags,
    INT iNumberOfPBDFlags,
    LONG lFlags,
    LPVOID pbcPara = NULL );

```

Table 2.24:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszExeName	Pointer to a string whose value is the name of the executable file to be created.
lpszIconName	Pointer to a string whose value is the name of an icon file. The icon file must already exist.
lpszPBRName	Pointer to a string whose value is the name of a PowerBuilder resource file. The resource file you name must already exist. If the application has no resource file, specify 0 for the pointer.
pLinkErrProc	Pointer to the <code>PBORCA_ExecutableCreate</code> callback function. The callback function is called for each link error that occurs.

Argument	Description
	<p>The information ORCA passes to the callback function is the message text, stored in a structure of type <code>PBORCA_LINKERR</code>.</p> <p>If you don't want to use a callback function, set <code>pLinkErrProc</code> to 0.</p>
<code>pUserData</code>	<p>Pointer to user data to be passed to the <code>PBORCA_ExecutableCreate</code> callback function.</p> <p>The user data typically includes the buffer or a pointer to the buffer in which the callback function formats the directory information as well as information about the size of the buffer.</p> <p>If you are not using a callback function, set <code>pUserData</code> to 0.</p>
<code>iPBDFlags</code>	<p>Pointer to an array of integers that indicate which libraries on the ORCA session's library list should be built into PowerBuilder dynamic libraries (PBDs). Each array element corresponds to a library in the library list. Flag values are:</p> <ul style="list-style-type: none"> • 0 -- Include the library's objects in the executable file • 1 -- The library is already a PBD or PowerBuilder DLL and its objects should not be included in the executable
<code>iNumberOfPBDFlags</code>	<p>The number of elements in the array <code>iPBDFlags</code>, which should be the same as the number of libraries on ORCA's library list.</p>
<code>IFlags</code>	<p>A long value whose value indicates which code generation options to apply when building the executable.</p> <p>Setting <code>IFlags</code> to 0 generates a native Pcode executable. Additional settings for machine code are described in Usage below.</p> <p>Setting <code>IFlags</code> to <code>PBORCA_X64</code> generates a 64-bit executable.</p>
<code>pbcPara</code>	<p>Reserved for internal use. Always set <code>pbcPara</code> to NULL.</p>

Return value

INT. Typical return codes are:

Table 2.25:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-5 PBORCA_LIBLISTNOTSET	Library list not set
-12 PBORCA_LINKERROR	Link error
-13 PBORCA_CURRAPPLNOTSET	Current application not set

Usage

You must set the library list and current Application object before calling this function.

For more information about various options for building executables, see the PowerBuilder User's Guide.

Libraries used in the executable

The executable being built incorporates the objects in the libraries on ORCA's library list. The library list must be set by calling `PBORCA_SessionSetLibraryList` before creating an executable.

The `iPBDFlags` argument lets you specify which libraries are PBDs and which will be built into the executable file. In the `iPBDFlags` array, each integer is associated with a library on ORCA's library list. When you set an integer to 1, the objects in the corresponding library are already built into a PBD file (if you are generating Pcode) or a PowerBuilder DLL (if you are generating machine code). Objects in libraries whose integer flag is set to 0 will be built into the main executable file.

Before you call `PBORCA_ExecutableCreate`, you must call `PBORCA_DynamicLibraryCreate` to create the PBDs or DLLs that you identify in the `iPBDFlags` array.

Setting code generation options

In the `IFlags` argument, you can set various machine code generation options by setting individual bits. The following table shows what each defined bit means in the long value and what constants to use in a bitwise OR expression to set the option. Bits not listed are reserved.

Table 2.26:

Bit	Value and meaning	Constant to include in ORed expression
0	0 = Pcode 1 = Machine code	To get machine code, use <code>PBORCA_MACHINE_CODE</code> or <code>PBORCA_MACHINE_CODE_NATIVE</code>
1	0 = Native code 1 = 16-bit code	To get 16-bit machine code, use <code>PBORCA_MACHINE_CODE</code>

Bit	Value and meaning	Constant to include in ORed expression
		<p>and PBORCA_MACHINE_CODE_16</p> <p>To get 16-bit Pcode, use PBORCA_P_CODE_16</p> <p>Not supported after PowerBuilder 7</p> <p>PowerBuilder no longer supports the Windows 3.x 16- bit platform.</p>
2	0 = No Open Server 1 = Open Server	<p>To build an Open Server executable, use PBORCA_OPEN_SERVER</p> <p>Not supported after PowerBuilder 5</p> <p>The OpenClientServer driver was no longer supported after PowerBuilder 5. Therefore, the Open Server executable option is no longer supported.</p>
4	0 = No trace information 1 = Trace information	<p>To get trace information, use PBORCA_TRACE_INFO</p>
5	0 = No error context 1 = Error context	<p>To get error context information, use PBORCA_ERROR_CONTEXT</p> <p>Error context provides the script name and line number of an error.</p>
8	0 = No optimization 1 = Optimization	<p>See Bit 9</p>
9	0 = Optimize for speed 1 = Optimize for space	<p>To optimize the executable for speed, use PBORCA_MACHINE_CODE_OPT or PBORCA_MACHINE_CODE_OPT_SPEED</p> <p>To optimize the executable for space, use PBORCA_MACHINE_CODE_OPT and PBORCA_MACHINE_CODE_OPT_SPACE</p>
10	0 = Old style visual controls	PBORCA_NEW_VISUAL_STYLE_CONTR

Bit	Value and meaning	Constant to include in ORed expression
	1 = New style visual controls (XP)	
12	1 = PocketBuilder desktop	PBORCA_PK_DESKTOP (Obsolete)
13	1 = PocketBuilder ARM	PBORCA_PK_PPCARM (Obsolete)
14	1 = PocketBuilder EM86	PBORCA_PK_PPCEM86 (Obsolete)
15	1 = PocketBuilder X86	PBORCA_PK_PPCX86 (Obsolete)
16	1 = PocketBuilder Smartphone ARM	PBORCA_PK_SPHONEARM (Obsolete)
17	1 = PocketBuilder Smartphone X86	PBORCA_PK_SPHONEX86 (Obsolete)

To generate Pcode, IFlags must be 0. The other bits are not relevant:

```
IFlags = PBORCA_P_CODE;
```

To set the IFlags argument for various machine-code options, the bit flag constants are ORed together to get the combination you want:

```
IFlags = PBORCA_MACHINE_CODE |  
         PBORCA_MACHINE_CODE_OPT |  
         PBORCA_MACHINE_CODE_OPT_SPACE;
```

Constants are defined in PBORCA.H for typical option combinations. They are:

PBORCA_MACHINE_DEFAULT

Meaning native machine code optimized for speed

Equivalent to:

```
PBORCA_MACHINE_CODE |  
         PBORCA_MACHINE_CODE_OPT_SPEED
```

PBORCA_MACHINE_DEBUG

Meaning native machine code with trace information and error context information

Equivalent to:

```
PBORCA_MACHINE_CODE | PBORCA_TRACE_INFO |  
         PBORCA_ERROR_CONTEXT
```

eClobber setting

If the executable file already exists in the file system, the current setting of the eClobber property in the ORCA configuration block (that you set with a `PBORCA_ConfigureSession` call) determines whether `PBORCA_ExecutableCreate` succeeds or fails.

Table 2.27:

Current eClobber setting	PBORCA_ExecutableCreate
PBORCA_NOCLOBBER or PBORCA_CLOBBER_DECIDED_BY_SYSTEM	FAILS when an executable file already exists in the file system, regardless of the file attribute settings
PBORCA_CLOBBER	Succeeds when the existing executable file has read-write attributes; fails when the executable file has read-only attributes
PBORCA_CLOBBER_ALWAYS	Succeeds regardless of the file attribute settings of an existing executable file

Examples

This example builds a native machine code executable optimized for speed using ORCA's library list and current application. Suppose that the current ORCA session has a library list with four entries. The example generates DLLs for the last two libraries.

The callback function is called `LinkErrors`, and `lpUserData` points to an empty buffer to be populated by the callback function:

```
LPTSTR pszExecFile;
LPTSTR pszIconFile;
LPTSTR pszResourceFile;
int iPBDFlags[4];
long lBuildOptions;
int rtn;

fpLinkProc = (PBORCA_LNKPROC) LinkProc;
// specify file names
pszExecFile = _TEXT("c:\\app\\process.exe");
pszIconFile = _TEXT("c:\\app\\process.ico");
pszResourceFile = _TEXT("c:\\app\\process.pbr");

iPBDFlags[0] = 0;
iPBDFlags[1] = 0;
iPBDFlags[2] = 1;
iPBDFlags[3] = 1;

lBuildOptions = PBORCA_MACHINE_CODE_NATIVE |
    PBORCA_MACHINE_CODE_OPT_SPEED;

// create executable
rtn = PBORCA_ExecutableCreate(
    lpORCA_Info->hORCASession,
    pszExecFile, pszIconFile, pszResourceFile,
    fpLinkProc, lpUserData,
    (INT FAR *) iPBDFlags, 4, lBuildOptions, NULL );
```

For more information about setting up the data buffer for the callback, see [Content of a callback function](#) and the example for [PBORCA_LibraryDirectory](#).

In these examples, session information is saved in the data structure `ORCA_Info`, shown in [About the examples](#).

See also

[PBORCA_ConfigureSession](#)

[PBORCA_DynamicLibraryCreate](#)

2.14 PBORCA_LibraryCommentModify

Description

Modifies the comment for a PowerBuilder library.

Syntax

```
INT PBORCA_LibraryCommentModify ( HPBORCA hORCASession,
    LPTSTR lpszLibName,
    LPTSTR lpszLibComments );
```

Table 2.28:

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibName	Pointer to a string whose value is the name of the library whose comments you want to change
lpszLibComments	Pointer to a string whose value is the new library comments

Return value

INT. Typical return codes are:

Table 2.29:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-3 PBORCA_OBJNOTFOUND	Library not found
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error

Usage

You don't need to set the library list or current application before calling this function.

Examples

This example changes the comments for the library MASTER.PBL:

```
LPTSTR pszLibraryName;
LPTSTR pszLibraryComments;
// Specify library name and comment string
pszLibraryName =
    _TEXT("c:\\appeon\\pb2019\\demo\\master.pbl");
pszLibraryComments =
    _TEXT("PBL contains ancestor objects for XYZ app.");
// Insert comments into library
```

```
lpORCA_Info->lReturnCode =
    PBORCA_LibraryCommentModify(
        lpORCA_Info->hORCASession,
        pszLibraryName, pszLibraryComments);
```

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA_LibraryCreate](#)

2.15 PBORCA_LibraryCreate

Description

Creates a new PowerBuilder library.

Syntax

```
INT PBORCA_LibraryCreate ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszLibraryComments );
```

Table 2.30:

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the file name of the library to be created
lpszLibraryComments	Pointer to a string whose value is a comment documenting the new library

Return value

INT. Typical return codes are:

Table 2.31:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error
-8 PBORCA_OBJEXISTS	Object already exists
-9 PBORCA_INVALIDNAME	Library name is not valid

Usage

You do not need to set the library list or current application before calling this function.

Adding objects

PBORCA_LibraryCreate creates an empty library file on disk. You can add objects to the library from other libraries with functions like PBORCA_LibraryEntryCopy and PBORCA_CheckOutEntry. If you set the library list so that it includes the new library and then set the current application, you can import object source code with PBORCA_CompileEntryImport and PBORCA_CompileEntryImportList.

Examples

This example creates a library called NEWLIB.PBL and provides a descriptive comment:

```
LPTSTR pszLibraryName;
LPTSTR pszLibraryComments;
// Specify library name and comment string
pszLibraryName =
    _TEXT("c:\\apeon\\pb2019\\demo\\newlib.pbl");
pszLibraryComments =
    _TEXT("PBL contains ancestor objects for XYZ app.");
// Create the library
lpORCA_Info->lReturnCode =
    PBORCA_LibraryCreate(lpORCA_Info->hORCASession,
        pszLibraryName, pszLibraryComments);
```

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA_LibraryDelete](#)

2.16 PBORCA_LibraryDelete

Description

Deletes a PowerBuilder library file from disk.

Syntax

```
INT PBORCA_LibraryDelete ( HPBORCA hORCASession,
    LPTSTR
    lpszLibraryName );
```

Table 2.32:

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the file name of the library to be deleted

Return value

INT. Typical return codes are:

Table 2.33:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list

Return code	Description
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error

Usage

You do not need to set the library list or current application before calling this function. You must set the eClobber configuration property to PBORCA_CLOBBER_ALWAYS if you want to delete a PowerBuilder library that has a read-only attribute.

Examples

This example deletes a library called EXTRA.PBL:

```
LPTSTR pszLibraryName;
// Specify library name
pszLibraryName =
    _TEXT("c:\\apeon\\pb2019\\demo\\extra.pbl");

// Delete the Library
lpORCA_Info->lReturnCode =
    PBORCA_LibraryDelete(lpORCA_Info->hORCA_Session,
        pszLibraryName);
```

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA_ConfigureSession](#)

[PBORCA_LibraryCreate](#)

2.17 PBORCA_LibraryDirectory

Description

Reports information about the directory of a PowerBuilder library, including the list of objects in the directory.

Syntax

```
INT PBORCA_LibraryDirectory ( HPBORCA hORCA_Session,
    LPTSTR lpszLibName,
    LPTSTR lpszLibComments,
    INT iCmntsBuffLen,
    PBORCA_LISTPROC pListProc,
    LPVOID pUserData );
```

Table 2.34:

Argument	Description
hORCA_Session	Handle to previously established ORCA session.
lpszLibName	Pointer to a string whose value is the file name of the library for which you want directory information.

Argument	Description
lpszLibComments	Pointer to a buffer in which ORCA will put comments stored with the library.
iCmntsBuffLen	Length of the buffer (specified in TCHARs) pointed to by lpszLibComments. The recommended length is PBORCA_MAXCOMMENTS + 1.
pListProc	Pointer to the PBORCA_LibraryDirectory callback function. The callback function is called for each entry in the library. The information ORCA passes to the callback function is entry name, comments, size of entry, and modification time, stored in a structure of type PBORCA_DIRENTRY.
pUserData	Pointer to user data to be passed to the PBORCA_LibraryDirectory callback function. The user data typically includes the buffer or a pointer to the buffer in which the callback function formats the directory information as well as information about the size of the buffer.

Return value

INT. Typical return codes are:

Table 2.35:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error

Usage

You do not need to set the library list or current application before calling this function.

Comments for the library

PBORCA_LibraryDirectory puts the library comments in the string pointed to by lpszLibComments. The callback function can store comments for individual objects in the UserData buffer.

Information about library entries

The information you get back about the individual entries in the library depends on the processing you provide in the callback function. ORCA passes information to the callback function about a library entry in the structure PBORCA_DIRENTRY. The callback function

can examine that structure and store any information it wants in the buffer pointed to by `pUserData`.

When you call `PBORCA_LibraryDirectory`, you do not know how many entries there are in the library. There are two approaches you can take:

- Allocate a reasonably sized block of memory and reallocate the buffer if it overflows (illustrated in [About ORCA callback functions](#)).
- Let `lpUserDataBuffer` point to the head of a linked list. For each `PBORCA_DIRENTRY` returned, dynamically allocate a new list entry to capture the required information (illustrated in the example that follows).

Examples

This example defines a linked list header:

```
typedef struct libinfo_head
{
    TCHAR        szLibName[PBORCA_SCC_PATH_LEN];
    TCHAR        szComments[PBORCA_MAXCOMMENT+1];
    INT          iNumEntries;
    PLIBINFO_ENTRY    pEntryAnchor;
    PLIBINFO_ENTRY    pLast;
} LIBINFO_HEAD, FAR *PLIBINFO_HEAD;
```

Each invocation of the `DirectoryProc` callback function allocates a new linked list entry, defined as follows:

```
typedef struct libinfo_entry
{
    TCHAR        szEntryName[41];
    LONG         lEntrySize;
    LONG         lObjectSize;
    LONG         lSourceSize;
    PBORCA_TYPE  otEntryType;
    libinfo_entry * pNext;
} LIBINFO_ENTRY, FAR *PLIBINFO_ENTRY;

PBORCA_LISTPROC    fpDirectoryProc;
PLIBINFO_HEAD      pHead;
fpDirectoryProc    = (PBORCA_LISTPROC) DirectoryProc;
pHead = new LIBINFO_HEAD;
_tcscpy(pHead->szLibName, _TEXT("c:\\myapp\\test.pbl"));
memset(pHead->szComments, 0x00,
    sizeof(pHead->szComments));
pHead->iNumEntries = 0;
pHead->pEntryAnchor = NULL;
pHead->pLast = NULL;
lpORCA_Info->lReturnCode = PBORCA_LibraryDirectory(
    lpORCA_Info->hORCASession,
    pHead->szLibName,
    pHead->szComments,
    (PBORCA_MAXCOMMENT+1), // specify length in TCHARs
    fpDirectoryProc,
    pHead);
// See PBORCA_LibraryEntryInformation example
if (lpORCA_Info->lReturnCode == PBORCA_OK)
    GetEntryInfo(pHead);
Cleanup(pHead);
// Cleanup - Release allocated memory
INT Cleanup(PLIBINFO_HEAD pHead)
```

```

{
    INT          iErrCode = PBORCA_OK;
    PLIBINFO_ENTRY    pCurrEntry;
    PLIBINFO_ENTRY    pNext;
    INT          idx;
for (idx = 0, pCurrEntry = pHead->pEntryAnchor;
    (idx < pHead->iNumEntries) && pCurrEntry; idx++)
{
    pNext = pCurrEntry->pNext;
    delete pCurrEntry;
    if (pNext)
        pCurrEntry = pNext;
    else pCurrEntry = NULL;
}
delete pHead;
return iErrCode;
}
// Callback procedure used by PBORCA_LibraryDirectory
void __stdcall DirectoryProc(PBORCA_DIRENTRY
    *pDirEntry, LPVOID lpUserData)
{
    PLIBINFO_HEAD    pHead;
    PLIBINFO_ENTRY    pNewEntry;
    PLIBINFO_ENTRY    pTemp;

    pHead = (PLIBINFO_HEAD) lpUserData;
    pNewEntry = (PLIBINFO_ENTRY) new LIBINFO_ENTRY;
    memset(pNewEntry, 0x00, sizeof(LIBINFO_ENTRY));
    if (pHead->iNumEntries == 0)
    {
        pHead->pEntryAnchor = pNewEntry;
        pHead->pLast = pNewEntry;
    }
    else
    {
        pTemp = pHead->pLast;
        pTemp->pNext = pNewEntry;
        pHead->pLast = pNewEntry;
    }
    pHead->iNumEntries++;
    _tcscpy(pNewEntry->szEntryName,
        pDirEntry->lpszEntryName);
    pNewEntry->lEntrySize = pDirEntry->lEntrySize;
    pNewEntry->otEntryType = pDirEntry->otEntryType;
}

```

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA_LibraryEntryInformation](#)

2.18 PBORCA_LibraryEntryCopy

Description

Copies a PowerBuilder library entry from one library to another.

Syntax

```

INT PBORCA_LibraryEntryCopy ( HPBORCA hORCASession,
    LPTSTR lpszSourceLibName,
    LPTSTR lpszDestLibName,
    LPTSTR lpszEntryName,

```

```
PBORCA_TYPE otEntryType );
```

Table 2.36:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszSourceLibName	Pointer to a string whose value is the file name of the source library containing the object.
lpszDestLibName	Pointer to a string whose value is the file name of the destination library to which you want to copy the object.
lpszEntryName	Pointer to a string whose value is the name of the object being copied.
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being copied. Values are: PBORCA_APPLICATION PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT

Return value

INT. Typical return codes are:

Table 2.37:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-3 PBORCA_OBJNOTFOUND	Object not found
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error

Usage

You do not need to set the library list or current application before calling this function.

Unlike `PBORCA_CompileEntryImport`, which requires two separate API calls, `PBORCA_LibraryEntryCopy` automatically copies the source component and then copies the binary component of an object if it is present.

Examples

This example copies a DataWindow named `d_labels` from the library `SOURCE.PBL` to `DESTIN.PBL`:

```
lpORCA_Info->lReturnCode = PBORCA_LibraryEntryCopy(
    lpORCA_Info->hORCASession,
    _TEXT("c:\\app\\source.pbl"),
    _TEXT("c:\\app\\destin.pbl"),
    _TEXT("d_labels"), PBORCA_DATAWINDOW);
```

This example assumes that the pointers for `lpSourceLibraryName`, `lpDestinationLibraryName`, and `lpEntryName` point to valid library and object names and that `otEntryType` is a valid object type:

```
lpORCA_Info->lReturnCode = PBORCA_LibraryEntryCopy(
    lpORCA_Info->hORCASession,
    lpSourceLibraryName,
    lpDestinationLibraryName,
    lpEntryName, otEntryType );
```

See also

[PBORCA_LibraryDelete](#)

[PBORCA_LibraryEntryMove](#)

2.19 PBORCA_LibraryEntryDelete

Description

Deletes a PowerBuilder library entry.

Syntax

```
INT PBORCA_LibraryEntryDelete (HPBORCA hORCASession,
    LPTSTR lpzLibName,
    LPTSTR lpzEntryName,
    PBORCA_TYPE otEntryType );
```

Table 2.38:

Argument	Description
<code>hORCASession</code>	Handle to previously established ORCA session.
<code>lpzLibName</code>	Pointer to a string whose value is the file name of the library containing the object.
<code>lpzEntryName</code>	Pointer to a string whose value is the name of the object being deleted.
<code>otEntryType</code>	A value of the <code>PBORCA_TYPE</code> enumerated data type specifying the object type of the entry being deleted. Values are:

Argument	Description
	PBORCA_APPLICATION
	PBORCA_DATAWINDOW
	PBORCA_FUNCTION
	PBORCA_MENU
	PBORCA_QUERY
	PBORCA_STRUCTURE
	PBORCA_USEROBJECT
	PBORCA_WINDOW
	PBORCA_PIPELINE
	PBORCA_PROJECT
	PBORCA_PROXYOBJECT

Return value

INT. Typical return codes are:

Table 2.39:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-3 PBORCA_OBJNOTFOUND	Object not found
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error

Usage

You do not need to set the library list or current application before calling this function.

Examples

This example deletes a DataWindow named d_labels from the library SOURCE.PBL:

```
l rtn = PBORCA_LibraryEntryDelete(
    lpORCA_Info->hORCA_Session,
    _TEXT("c:\\app\\source.pbl"),
    _TEXT("d_labels"), PBORCA_DATAWINDOW);
```

This example assumes that the pointers lpszLibraryName and lpszEntryName point to valid library and object names and that otEntryType is a valid object type:

```
lpORCA_Info->lReturnCode = PBORCA_LibraryEntryDelete(
    lpORCA_Info->hORCA_Session,
    lpszLibraryName,
    lpszEntryName,
    otEntryType);
```

See also

[PBORCA_LibraryEntryCopy](#)

[PBORCA_LibraryEntryMove](#)

2.20 PBORCA_LibraryEntryExport

Description

Exports the source code for a PowerBuilder library entry to a source buffer or file.

Syntax

```
INT PBORCA_LibraryEntryExport ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszEntryName,
    PBORCA_TYPE otEntryType,
    LPTSTR lpszExportBuffer,
    LONG lExportBufferSize );
```

Table 2.40:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	Pointer to a string whose value is the file name of the library containing the object you want to export.
lpszEntryName	Pointer to a string whose value is the name of the object being exported.
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being exported. Values are: PBORCA_APPLICATION PBORCA_BINARY PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW
lpszExportBuffer	Pointer to the data buffer in which ORCA stores the code for the exported source when

Argument	Description
	the PBORCA_CONFIG_SESSION property bExportCreateFile is FALSE. This argument can be NULL if bExportCreateFile is TRUE.
lExportBufferSize	Size in bytes of lpszExportBuffer. This argument is not required if the PBORCA_CONFIG_SESSION property bExportCreateFile is TRUE.

Return value

INT. Typical return codes are:

Table 2.41:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-3 PBORCA_OBJNOTFOUND	Object not found
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error
-10 PBORCA_BUFFERTOOSMALL	Buffer size is too small
-33 PBORCA_DBCSERROR	Locale setting error when converting Unicode to ANSI_DBCS

Usage

You do not need to set the library list or current application before calling this function.

Changes for PowerBuilder 10 and higher

In PowerBuilder 10 and higher, you can customize behavior of this function using PBORCA_CONFIG_SESSION variables. However, for backward compatibility, the default behavior has not changed.

How the source code is returned

If pConfigSession->bExportCreateFile is FALSE, the object's source code is returned in the export buffer. If the bExportCreateFile property is TRUE, the source is written to a file in the directory pointed to by pConfigSession->pExportDirectory.

If pConfigSession->bExportHeaders is TRUE, ORCA writes the two export header lines to the beginning of the export buffer or file. The exported source code includes carriage return (hex 0D) and new line (hex 0A) characters at the end of each display line.

Source code encoding

PowerBuilder exports source in four different encoding formats. By default, ANSI/DBCS clients export source in PBORCA_ANSI_DBCS format; Unicode clients export source in

PBORCA_UNICODE format. You can explicitly request an encoding format by setting `pConfigSession->eExportEncoding`.

Binary component

In PowerBuilder, you can explicitly request that the binary component of an object be included automatically in the export buffer or file by setting `pConfigSession->eExportIncludeBinary = TRUE`.

This is the recommended setting for new development. Because previous releases of ORCA did not support this feature, the old technique is still supported.

Denigrated technique

As in previous versions, after each `PBORCA_LibraryEntryExport` request, you can call `PBORCA_LibraryEntryInformation` with an `otEntryType` of `PBORCA_BINARY`. This function returns `PBORCA_OK` when binary data exists and you could make a second `PBORCA_LibraryEntryExport` call with `otEntryType` set to `PBORCA_BINARY`. For backward compatibility, setting `otEntryType` to `PBORCA_BINARY` causes the following configuration properties to be ignored: `pConfigSession->bExportHeaders = TRUE` and `pConfigSession->bExportIncludeBinary = TRUE`.

Size of source code

To find out the size of the source for an object before calling the export function, call the `PBORCA_LibraryEntryInformation` function first and use the `pEntryInfo->lSourceSize` information to calculate an appropriate `lExportBufferSize` value. `lExportBufferSize` is the size of `lpszExportBuffer` represented in bytes.

ORCA export processing performs all necessary data conversions before determining whether the allocated buffer is large enough to contain the export source. If not, it returns a `PBORCA_BUFFERTOOSMALL` return code. If `lExportBufferSize` is exactly the required length, `PBORCA_LibraryEntryExport` succeeds, but does not append a null terminator to the exported source. If `lExportBufferSize` is sufficiently large, ORCA appends a null terminator. Apeon recommends allocating a buffer sufficiently large to accommodate data conversions and a null terminator. `lExportBufferSize` is ignored if `pConfigSession->bExportCreateFile = TRUE`.

Determining the source size after data conversion and export

If you need to know the size of the actual buffer or file returned, you can call `PBORCA_LibraryEntryExportEx` instead of `PBORCA_LibraryEntryExport`. These functions behave exactly alike except that the `PBORCA_LibraryEntryExportEx` function signature includes an additional `*plReturnSize` argument.

Overwriting existing export files

The value of `pConfigSession->eClobber` determines whether existing export files are overwritten. If the export files do not exist, `PBORCA_LibraryEntryExport` returns `PBORCA_OK` regardless of the `eClobber` setting. The following table shows how the `eClobber` setting changes the action of `PBORCA_LibraryEntryExport` when export files already exist. A return value of `PBORCA_OBJEXISTS` means that the existing files were not overwritten.

Table 2.42:

PConfigSession->eClobber setting	Return value if read/write file exists	Return value if read-only file exists
PBORCA_NOCLOBBER	PBORCA_OBJEXISTS	PBORCA_OBJEXISTS
PBORCA_CLOBBER	PBORCA_OK	PBORCA_OBJEXISTS
PBORCA_CLOBBER_ALWAYS	PBORCA_OK	PBORCA_OK
PBORCA_CLOBBER_DECIDE	PBORCA_OBJEXISTS	PBORCA_OBJEXISTS

Examples

This example exports a DataWindow named d_labels from the library SOURCE.PBL. It puts the PBORCA_UTF8 source code in a buffer called szEntrySource. Export headers are included:

```
TCHAR szEntrySource[60000];
// Indicate UTF8 source encoding
lpORCA_Info->pConfig->eExportEncoding = PBORCA_UTF8;
// Request export headers
lpORCA_Info->pConfig->bExportHeaders = TRUE;
// Write output to memory buffer
lpORCA_Info->pConfig->bExportCreateFile = FALSE;
// Override existing session configuration
PBORCA_ConfigureSession(lpORCA_Info->hORCASession,
lpORCA_Info->pConfig);
lpORCA_Info->lReturnCode = PBORCA_LibraryEntryExport(
lpORCA_Info->hORCASession,
TEXT("c:\\app\\source.pbl"),
TEXT("d_labels"), PBORCA_DATAWINDOW,
(LPTSTR) szEntrySource, 60000);
```

This example exports a DataWindow named d_labels from the library SOURCE.PBL. It writes the PBORCA_UNICODE source code to c:\app\d_labels.srd. Export headers are included:

```
// Indicate UNICODE source encoding
lpORCA_Info->pConfig->eExportEncoding = PBORCA_UNICODE;
// Write to file
lpORCA_Info->pConfig->bExportCreateFile = TRUE;
// Specify output directory
lpORCA_Info->pConfig->pExportDirectory = TEXT("c:\\app");
// Request export headers
lpORCA_Info->pConfig->bExportHeaders = TRUE;
// Override existing session configuration
PBORCA_ConfigureSession(lpORCA_Info->hORCASession,
lpORCA_Info->pConfig);
// Perform the actual export
lpORCA_Info->lReturnCode = PBORCA_LibraryEntryExport(
lpORCA_Info->hORCASession,
TEXT("c:\\app\\source.pbl"),
TEXT("d_labels"), PBORCA_DATAWINDOW,
NULL, 0);
```

This example exports a Window named w_connect from the library SOURCE.PBL. It contains an embedded OLE object. Both the source code and the binary object are exported to c:\app\w_connect.srw. Export headers are included and the source is written in PBORCA_ANSI_DBCS format:

```
// Indicate ANSI_DBCS source encoding
```

```

lpORCA_Info->pConfig->eExportEncoding = PBORCA_ANSI_DBCS;
// Export to a file
lpORCA_Info->pConfig->bExportCreateFile = TRUE;
// Specify output directory
lpORCA_Info->pConfig->pExportDirectory = _TEXT("c:\\app");
// Request export headers
lpORCA_Info->pConfig->bExportHeaders = TRUE;
// Include binary component
lpORCA_Info->pConfig->bExportIncludeBinary = TRUE;
// Override existing session configuration
PBORCA_ConfigureSession(lpORCA_Info->hORCASession,
lpORCA_Info->pConfig);
// Perform the actual export
lpORCA_Info->lReturnCode = PBORCA_LibraryEntryExport(
    lpORCA_Info->hORCASession,
    _TEXT("c:\\app\\source.pbl"),
    _TEXT("w_connect"), PBORCA_WINDOW,
    NULL, 0);

```

See also[PBORCA_ConfigureSession](#)[PBORCA_CompiledEntryImport](#)[PBORCA_LibraryEntryExportEx](#)

2.21 PBORCA_LibraryEntryExportEx

Description

Exports the source code for a PowerBuilder library entry to a text buffer.

Syntax

```

INT PBORCA_LibraryEntryExportEx ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszEntryName,
    PBORCA_TYPE otEntryType,
    LPTSTR lpszExportBuffer,
    LONG lExportBufferSize
    LONG *plReturnSize);

```

Table 2.43:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	Pointer to a string whose value is the file name of the library containing the object you want to export.
lpszEntryName	Pointer to a string whose value is the name of the object being exported.
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being exported. Values are: PBORCA_APPLICATION

Argument	Description
	PBORCA_BINARY PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW
lpzExportBuffer	Pointer to the data buffer in which ORCA stores the code for the exported source when the PBORCA_CONFIG_SESSION property bExportCreateFile is FALSE. This argument can be NULL if bExportCreateFile is TRUE.
lExportBufferSize	Size in bytes of lpzExportBuffer. This argument is not required if the PBORCA_CONFIG_SESSION property bExportCreateFile is TRUE.
*plReturnSize	The size, in BYTES, of the exported source buffer or file.

Return value

INT. Typical return codes are:

Table 2.44:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-3 PBORCA_OBJNOTFOUND	Object not found
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error
-10 PBORCA_BUFFERTOOSMALL	Buffer size is too small
-33 PBORCA_DBCSERROR	Locale setting error when converting Unicode to ANSI_DBCS

Usage

This function behaves exactly like `PBORCA_LibraryEntryExport`, except that with `PBORCA_LibraryEntryExportEx`, the size of the exported source is returned to the caller in the additional `*plReturnSize` argument.

See also

[PBORCA_ConfigureSession](#)

[PBORCA_CompileEntryImport](#)

[PBORCA_LibraryEntryExport](#)

2.22 PBORCA_LibraryEntryInformation

Description

Returns information about an object in a PowerBuilder library. Information includes comments, size of source, size of object, and modification time.

Syntax

```
INT PBORCA_LibraryEntryInformation ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszEntryName,
    PBORCA_TYPE otEntryType,
    PPBORCA_ENTRYINFO pEntryInformationBlock );
```

Table 2.45:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	Pointer to a string whose value is the file name of the library containing the object for which you want information.
lpszEntryName	Pointer to a string whose value is the name of the object for which you want information.
otEntryType	A value of the <code>PBORCA_TYPE</code> enumerated data type specifying the object type of the entry. Values are: <code>PBORCA_APPLICATION</code> <code>PBORCA_DATAWINDOW</code> <code>PBORCA_FUNCTION</code> <code>PBORCA_MENU</code> <code>PBORCA_QUERY</code> <code>PBORCA_STRUCTURE</code> <code>PBORCA_USEROBJECT</code> <code>PBORCA_WINDOW</code> <code>PBORCA_PIPELINE</code>

Argument	Description
	PBORCA_PROJECT PBORCA_PROXYOBJECT PBORCA_BINARY
pEntryInformationBlock	Pointer to PBORCA_ENTRYINFO structure in which ORCA will store the requested information (see Usage below).

Return value

INT. Typical return codes are:

Table 2.46:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-3 PBORCA_OBJNOTFOUND	Object not found
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error

Usage

You do not need to set the library list or current application before calling this function.

How entry information is returned

PBORCA_LibraryEntryInformation stores information about an entry in the following structure. You pass a pointer to the structure in the pEntryInformationBlock argument:

```
typedef struct PBORCA_EntryInfo
{
    TCHAR szComments[PBORCA_MAXCOMMENT + 1];
    LONG lCreateTime; // time of entry create-mod
    LONG lObjectSize; // size of object in bytes
    LONG lSourceSize; // size of source in bytes
} PBORCA_ENTRYINFO, FAR *PPBORCA_ENTRYINFO;
```

Use for the source code size

PBORCA_LibraryEntryInformation is often used to estimate the size in bytes of the source buffer needed to obtain the export source of an object. The size of the exported source varies depending on the ConfigureSession settings in effect. The following table shows how ConfigureSession variables affect the lSourceSize value that LibraryEntryInformation returns:

Table 2.47:

ConfigureSession variable	Effect on lSourceSize
ANSI/DBCS ORCA client	No effect. User should calculate required buffer size based on the usage tips that follow this table.

ConfigureSession variable	Effect on lSourceSize
eExportEncoding	No effect. PBORCA_LibraryEntryInformation always returns the number of bytes required for Unicode source.
bExportHeaders=TRUE	If otEntryType is not PBORCA_BINARY, lSourceSize will be increased by the number of bytes needed to generate Unicode export headers.
bExportIncludeBinary=TRUE	If otEntryType is not PBORCA_BINARY, lSourceSize will be increased by the number of bytes needed to generate the Unicode representation of the binary object.

Calculating buffer size needed for non-Unicode encodings

The size of the buffer required for non-Unicode export encodings cannot be calculated in advance without actually performing the data transformation. Developers should make their own estimate to arrive at a reasonable buffer size to allocate. For example, if the source for an entry is entirely ANSI, simply divide the lSourceSize value by 2 and add 1 byte if you want a null terminator. For Unicode source, add 2 bytes for the null terminator.

Using PBORCA_BINARY for entry type

In previous releases of ORCA, it was necessary to call PBORCA_LibraryEntryInformation a second time with an otEntryType of PBORCA_BINARY to determine if an entry contained embedded OLE controls. This call determined the size of the buffer needed to hold the representation of the binary data to be exported. Although PowerBuilder still supports this feature for backward compatibility, it is more efficient to set pConfigSession->bExportIncludBinary = TRUE to obtain a buffer size sufficient for both the source and binary components of an entry.

Examples

This example obtains information about each object in a PBL. It is an extension of the example for [PBORCA_LibraryDirectory](#).

```

INT EntryInfo(PLIBINFO_HEAD pHead)
{
    INT iErrCode;
    INT idx;
    PLIBINFO_ENTRY pCurrEntry;
    PBORCA_ENTRYINFO InfoBlock;
    INT iErrCount = 0;
    for (idx = 0, pCurrEntry = pHead->pEntryAnchor;
        (idx < pHead->iNumEntries) && pCurrEntry;
        idx++, pCurrEntry = pCurrEntry->pNext)
    {
        iErrCode = PBORCA_LibraryEntryInformation(
            lpORCA_Info->hORCASession pHead->szLibName,
            pCurrEntry->szEntryName,
            pCurrEntry->otEntryType, &InfoBlock);

        if (iErrCode == PBORCA_OK)
    
```

```

{
    pCurrEntry->lSourceSize = InfoBlock.lSourceSize;
    pCurrEntry->lObjectSize = InfoBlock.lObjectSize;
}
else
{
    ErrorMsg();
    iErrCount++;
}
}
if (iErrCount)
    iErrCode = -1;
return iErrCode;
}

```

See also[PBORCA_LibraryDirectory](#)[PBORCA_LibraryEntryExport](#)

2.23 PBORCA_LibraryEntryMove

Description

Moves a PowerBuilder library entry from one library to another.

Syntax

```

INT PBORCA_LibraryEntryMove ( PBORCA hORCASession,
    LPTSTR lpszSourceLibName,
    LPTSTR lpszDestLibName,
    LPTSTR lpszEntryName,
    PBORCA_TYPE otEntryType );

```

Table 2.48:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszSourceLibName	Pointer to a string whose value is the file name of the source library containing the object.
lpszDestLibName	Pointer to a string whose value is the file name of the destination library to which you want to move the object.
lpszEntryName	Pointer to a string whose value is the name of the object being moved.
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being moved. Values are: PBORCA_APPLICATION PBORCA_DATAWINDOW PBORCA_FUNCTION

Argument	Description
	PBORCA_MENU
	PBORCA_QUERY
	PBORCA_STRUCTURE
	PBORCA_USEROBJECT
	PBORCA_WINDOW
	PBORCA_PIPELINE
	PBORCA_PROJECT
	PBORCA_PROXYOBJECT

Return value

INT. Typical return codes are:

Table 2.49:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-3 PBORCA_OBJNOTFOUND	Object not found
-4 PBORCA_BADLIBRARY	Bad library name
-7 PBORCA_LIBIOERROR	Library I/O error

Usage

You do not need to set the library list or current application before calling this function.

Like PBORCA_LibraryEntryCopy, one call to PBORCA_LibraryEntryMove automatically moves the source component and then moves the binary component of an object if it is present.

Examples

This example moves a DataWindow named d_labels from the library SOURCE.PBL to DESTIN.PBL:

```
lpORCA_Info->lReturnCode = PBORCA_LibraryEntryMove(
    lpORCA_Info->hORCASession,
    _TEXT("c:\\app\\source.pbl"),
    _TEXT("c:\\app\\destin.pbl"),
    _TEXT("d_labels"), PBORCA_DATAWINDOW);
```

This example assumes that the pointers for lpszSourceLibraryName, lpszDestinationLibraryName, and lpszEntryName point to valid library and object names and that otEntryType is a valid object type:

```
lpORCA_Info->lReturnCode = PBORCA_LibraryEntryMove(
    lpORCA_Info->hORCASession,
    lpszSourceLibraryName, lpszDestinationLibraryName,
    lpszEntryName, otEntryType );
```

See also[PBORCA_LibraryEntryCopy](#)[PBORCA_LibraryEntryDelete](#)**2.24 PBORCA_ObjectQueryHierarchy****Description**

Queries a PowerBuilder object to get a list of the objects in its ancestor hierarchy. Only windows, menus, and user objects have an ancestor hierarchy that can be queried.

Syntax

```
INT PBORCA_ObjectQueryHierarchy ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszEntryName,
    PBORCA_TYPE otEntryType,
    PBORCA_HIERPROC pHierarchyProc,
    LPVOID pUserData );
```

Table 2.50:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	Pointer to a string whose value is the file name of the library containing the object being queried.
lpszEntryName	Pointer to a string whose value is the name of the object being queried.
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being queried. The only values allowed are: PBORCA_WINDOW PBORCA_MENU PBORCA_USEROBJECT
pHierarchyProc	Pointer to the PBORCA_ObjectQueryHierarchy callback function. The callback function is called for each ancestor object. The information ORCA passes to the callback function is the ancestor object name, stored in a structure of type PBORCA_HIERARCHY.
pUserData	Pointer to user data to be passed to the PBORCA_ObjectQueryHierarchy callback function.

Argument	Description
	The user data typically includes the buffer or a pointer to the buffer in which the callback function stores the ancestor names as well as information about the size of the buffer.

Return value

INT. The return codes are:

Table 2.51:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-3 PBORCA_OBJNOTFOUND	Object not found
-4 PBORCA_BADLIBRARY	Bad library name
-5 PBORCA_LIBLISTNOTSET	Library list not set
-6 PBORCA_LIBNOTINLIST	Library not in library list
-7 PBORCA_LIBIOERROR	Library I/O error
-9 PBORCA_INVALIDNAME	Name does not follow PowerBuilder naming rules

Usage

You must set the library list and current Application object before calling this function.

Examples

This example queries the window object `w_processdata` in the library `WINDOWS.PBL` to get a list of its ancestors. The `lpUserData` buffer was previously set up to point to space for storing the list of names.

For each ancestor in the object's hierarchy, `PBORCA_ObjectQueryHierarchy` calls the callback `ObjectQueryHierarchy`. In the code you write for `ObjectQueryHierarchy`, you store the ancestor name in the buffer pointed to by `lpUserData`. In the example, the `lpUserData` buffer has already been set up:

```
PBORCA_HIERPROC fpHierarchyProc;
fpHierarchyProc = (PBORCA_HIERPROC)GetHierarchy;
lpORCA_Info->lReturnCode = PBORCA_ObjectQueryHierarchy(
    _TEXT("c:\\app\\windows.pbl"),
    _TEXT("w_processdata"),
    PBORCA_WINDOW,
    fpHierarchyProc,
    lpUserData );
```

For more information about setting up the data buffer for the callback, see [Content of a callback function](#) and the example for [PBORCA_LibraryDirectory](#).

In these examples, session information is saved in the data structure `ORCA_Info`, shown in [About the examples](#).

See also[PBORCA_ObjectQueryReference](#)**2.25 PBORCA_ObjectQueryReference****Description**

Queries a PowerBuilder object to get a list of its references to other objects.

Syntax

```
INT PBORCA_ObjectQueryReference ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszEntryName,
    PBORCA_TYPE otEntryType,
    PBORCA_REFPROC pRefProc,
    LPVOID pUserData );
```

Table 2.52:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszLibraryName	Pointer to a string whose value is the file name of the library containing the object being queried.
lpszEntryName	Pointer to a string whose value is the name of the object being queried.
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being queried. Values are: PBORCA_APPLICATION PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT
pRefProc	Pointer to the PBORCA_ObjectQueryReference callback function. The callback function is called for each referenced object.

Argument	Description
	The information ORCA passes to the callback function is the referenced object name, its library, and its object type, stored in a structure of type <code>PBORCA_REFERENCE</code> .
<code>pUserData</code>	<p>Pointer to user data to be passed to the <code>PBORCA_ObjectQueryReference</code> callback function.</p> <p>The user data typically includes the buffer or a pointer to the buffer in which the callback function stores the object information as well as information about the size of the buffer.</p>

Return value

INT. Typical return codes are:

Table 2.53:

Return code	Description
0 <code>PBORCA_OK</code>	Operation successful
-1 <code>PBORCA_INVALIDPARMS</code>	Invalid parameter list
-3 <code>PBORCA_OBJNOTFOUND</code>	Object not found
-4 <code>PBORCA_BADLIBRARY</code>	Bad library name
-5 <code>PBORCA_LIBLISTNOTSET</code>	Library list not set
-6 <code>PBORCA_LIBNOTINLIST</code>	Library not in library list
-9 <code>PBORCA_INVALIDNAME</code>	Name does not follow PowerBuilder naming rules

Usage

You must set the library list and current Application object before calling this function.

Examples

This example queries the window object `w_processdata` in the library `WINDOWS.PBL` to get a list of its referenced objects. For each object that `w_processdata` references, `PBORCA_ObjectQueryReference` calls the callback `ObjectQueryReference`. In the code you write for `ObjectQueryReference`, you store the object name in the buffer pointed to by `lpUserData`. In the example, the `lpUserData` buffer has already been set up:

```

PBORCA_REFPROC      fpRefProc;
fpRefProc = (PBORCA_REFPROC) GetReferences;
lpORCA_Info->lReturnCode = PBORCA_ObjectQueryReference(
    lpORCA_Info->hORCA_Session,
    _TEXT("c:\\app\\windows.pbl"),
    _TEXT("w_processdata"),
    PBORCA_WINDOW,
    fpRefProc,
    lpUserData );

```

For more information about setting up the data buffer for the callback, see [Content of a callback function](#) and the example for [PBORCA LibraryDirectory](#).

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA ObjectQueryHierarchy](#)

2.26 PBORCA_SccClose

Description

Closes the active SCC project.

Syntax

```
INT PBORCA_SccClose ( HPBORCA hORCASession );
```

Table 2.54:

Argument	Description
hORCASession	Handle to previously established ORCA session

Return value

INT.

Usage

This method calls SCCUninitialize to disconnect from the source control provider. Call PBORCA_SccClose before calling PBORCA_SessionClose.

See also

[PBORCA_SccConnect](#)

2.27 PBORCA_SccConnect

Description

Initializes source control and opens a project.

Syntax

```
INT PBORCA_SccConnect ( HPBORCA hORCASession, PBORCA_SCC *pConfig );
```

Table 2.55:

Argument	Description
hORCASession	Handle to previously established ORCA session
*pConfig	Pointer to a preallocated structure typically initialized to zeros

Return value

INT. Typical return codes are:

Table 2.56:

Return code	Description
0 PBORCA_OK	Operation successful
-22 PBORCA_SCCFAILURE	Could not connect to source control
-23 PBORCA_REGREADERERROR	Could not read registry
-24 PBORCA_LOADDLLFAILED	Could not load DLL
-25 PBORCA_SCCINITFAILED	Could not initialize SCC connection
-26 PBORCA_OPENPROJFAILED	Could not open project

Usage

This method initializes a source control session based on the connection information supplied in the PBORCA_SCC structure. The PBORCA_SCC structure is defined as follows:

```
typedef struct pborca_scc
{
    HWND hWnd;
    TCHAR szProviderName [PBORCA_SCC_NAME_LEN + 1];
    LONG *plCapabilities;
    TCHAR szUserID [PBORCA_SCC_USER_LEN + 1];
    TCHAR szProject [PBORCA_SCC_PATH_LEN + 1];
    TCHAR szLocalProjPath [PBORCA_SCC_PATH_LEN + 1];
    TCHAR szAuxPath [PBORCA_SCC_PATH_LEN + 1];
    TCHAR szLogFile [PBORCA_SCC_PATH_LEN + 1];
    LPTEXTOUTPROC pMsgHandler;
    LONG *pCommentLen;
    LONG lAppend;
    LPVOID pCommBlk;
} PBORCA_SCC;
```

You can either populate the structure manually or else call PBORCA_SccGetConnectProperties to obtain the connection information associated with a specific workspace file. This function:

- Opens the requested source control project
- Creates a CPB_OrcaSourceControl class that implements the PBORCA_SCC methods
- Defines a runtime environment that persists until PBORCA_SccClose is called

The runtime environment has four subsystems: runtime engine (rt), object manager (ob), PowerScript compiler (cm), and storage manager (stg). The runtime environment is used to process the target identified by a subsequent PBORCA_SccSetTarget call. To process multiple targets, you must close the SCC connection, close the ORCA session, and open a new ORCA session.

Examples

The following example connects to PBNative source control:

```

PBORCA_SCC          sccConfig;
memset(&sccConfig, 0x00, sizeof(PBORCA_SCC));
// Manually set up connection properties to PBNative
_tcscpy(sccConfig.szProviderName, _TEXT("PB Native"));
_tcscpy(sccConfig.szProject,
        _TEXT("c:\\PBNative_Archive\\qadb"));
_tcscpy(sccConfig.szUserID, _TEXT("Joe"));
_tcscpy(sccConfig.szLogFile, _TEXT("c:\\qadb\\orcasc.log"));
_tcscpy(sccConfig.szLocalProjPath, _TEXT("c:\\qadb"));
sccConfig.lAppend = 0;
lpORCA_Info->lReturnCode = PBORCA_SccConnect(
    lpORCA_Info->hORCASession,
    &sccConfig);

```

See also[PBORCA_SccClose](#)[PBORCA_SccConnectOffline](#)[PBORCA_SccGetConnectProperties](#)[PBORCA_SccSetTarget](#)

2.28 PBORCA_SccConnectOffline

Description

Opens a source-controlled project for refreshing and rebuilding offline.

Syntax

```

INT PBORCA_SccConnectOffline ( HPBORCA hORCASession,
    PBORCA_SCC *pConfig );

```

Table 2.57:

Argument	Description
hORCASession	Handle to previously established ORCA session
*pConfig	Pointer to a preallocated structure typically initialized to zeros

Return value

INT. Typical return codes are:

Table 2.58:

Return code	Description
0 PBORCA_OK	Operation successful
-22 PBORCA_SCCFAILURE	Could not connect to source control
-23 PBORCA_REGREADERERROR	Could not read registry
-24 PBORCA_LOADDLLFAILED	Could not load DLL
-25 PBORCA_SCCINITFAILED	Could not initialize SCC connection

Return code	Description
-26 PBORCA_OPENPROJFAILED	Could not open project

Usage

This function is applicable only when `PBORCA_SCC_IMPORTONLY` is specified on the subsequent `PBORCA_SccSetTarget` command.

Import-only processing assumes that all of the objects necessary to refresh a source-controlled target already exist on the local project path. Therefore, `PBORCA_SccConnectOffline` instantiates the ORCA source control class but does not actually connect to an SCC provider.

This function is particularly useful for developers who use laptop computers. While connected to the network, they can refresh their SCC client view. Then, during off hours, they can perform the time-consuming process of refreshing and rebuilding their application without the need for a network connection.

Examples

This example populates the `PBORCA_SCC` structure with connection information from the PocketBuilder `qadb.pkw` workspace file located in the current working directory. It then connects in offline mode and refreshes the `qadbtest.pbt` target that is located in the `qadbtest` subdirectory under the current working directory. Only objects that are out of sync will be refreshed. Objects checked out by the current user will not be overwritten:

```
PBORCA_SCC          sccConfig;
TCHAR              szWorkspace[PBORCA_SCC_PATH_LEN];
TCHAR              szTarget[PBORCA_SCC_PATH_LEN];
LONG               lFlags;
memset(&sccConfig, 0x00, sizeof(PBORCA_SCC));
_tcscpy(szWorkspace, _TEXT("qadb.pkw"));
lpORCA_Info->lReturnCode =
PBORCA_SccGetConnectProperties(
    lpORCA_Info->hORCASession,
    szWorkspace,
    &sccConfig);
if (lpORCA_Info->lReturnCode == PBORCA_OK)
{
    // Specify a different log file for the build operation
    _tcscpy(sccConfig.szLogFile, _TEXT("bldqadb.log"));
    sccConfig.lAppend = 0;
    lpORCA_Info->lReturnCode = PBORCA_SccConnectOffline(
        lpORCA_Info->hORCASession, &sccConfig);
    if (lpORCA_Info->lReturnCode == PBORCA_OK)
    {
        _tcscpy(szTarget, _TEXT("qadbtest\\qadbtest.pkt"));
        lFlags = PBORCA_SCC_IMPORTONLY |
            PBORCA_SCC_OUTOFDATE |
            PBORCA_SCC_EXCLUDE_CHECKOUT;
        lpORCA_Info->lReturnCode = PBORCA_SccSetTarget(
            lpORCA_Info->hORCASession,
            szTarget,
            lFlags,
            NULL,
            NULL);
    }
    if (lpORCA_Info->lReturnCode == PBORCA_OK)
    {
        lpORCA_Info->lReturnCode = PBORCA_SccRefreshTarget(
            lpORCA_Info->hORCASession, PBORCA_FULL_REBUILD);
    }
}
```

```
}
}
}
```

See also[PBORCA_SccClose](#)[PBORCA_SccConnect](#)[PBORCA_SccGetConnectProperties](#)[PBORCA_SccSetTarget](#)

2.29 PBORCA_SccExcludeLibraryList

Description

Names the libraries in the target library list that should not be synchronized in the next PBORCA_SccRefreshTarget operation.

Syntax

```
INT PBORCA_SccExcludeLibraryList ( HPBORCA hORCASession,
    LPTSTR *pLibNames,
    INT iNumberOfLibs );
```

Table 2.59:

Argument	Description
hORCASession	Handle to previously established ORCA session
*pLibNames	Names of the libraries not to be refreshed
iNumberOfLibs	Number of libraries not to be refreshed

Return value

INT.

Usage

This method is useful if PBLs are shared among multiple targets and you are certain that the libraries you list have been successfully refreshed by a previous PBORCA_SccRefreshTarget operation. The refresh target operation will not refresh the libraries that are excluded; however, the excluded libraries will still be used in the full rebuild of the application.

Examples

A previous PBORCA_SccRefreshTarget operation has successfully refreshed three of the four PocketBuilder libraries in this target library list.

```
LPTSTR    pExcludeArray[3];
INT       lExcludeCount = 3;
TCHAR     szTarget[PBORCA_SCC_PATH_LEN];
LONG      lFlags;
pExcludeArray[0] = new TCHAR[PBORCA_SCC_PATH_LEN];
pExcludeArray[1] = new TCHAR[PBORCA_SCC_PATH_LEN];
pExcludeArray[2] = new TCHAR[PBORCA_SCC_PATH_LEN];
_tcscpy(pExcludeArray[0],
TEXT("..\shared_obj\shared_obj.pk1"));
```

```

_tcscopy(pExcludeArray[1],
_TEXT("../datatypes\\datatypes.pkl"));
_tcscopy(pExcludeArray[2],
_TEXT("../chgreqs\\chgreqs.pkl"));
// Open ORCA Session, connect to SCC
// --
_tcscopy(szTarget, _TEXT("dbauto\\dbauto.pkt"));
lFlags = PBORCA_SCC_IMPORTONLY | PBORCA_SCC_OUTOFDATE |
        PBORCA_SCC_EXCLUDE_CHECKOUT;
lpORCA_Info->lReturnCode = PBORCA_SccSetTarget(
lpORCA_Info->hORCASession, szTarget, lFlags, NULL, NULL);

```

```

if (lpORCA_Info->lReturnCode == PBORCA_OK)
{
    lpORCA_Info->lReturnCode = PBORCA_SccExcludeLibraryList(
lpORCA_Info->hORCASession, pExcludeArray,
lExcludeCount);

if (lpORCA_Info->lReturnCode == PBORCA_OK)
{
    lpORCA_Info->lReturnCode = PBORCA_SccRefreshTarget(
lpORCA_Info->hORCASession, PBORCA_FULL_REBUILD );
}
}
for (int i = 0; i < lExcludeCount; i++)
delete [] pExcludeArray[i];

```

See also[PBORCA_SccRefreshTarget](#)[PBORCA_SccSetTarget](#)

2.30 PBORCA_SccGetConnectProperties

Description

Returns the SCC connection properties associated with a PowerBuilder workspace.

Syntax

```

INT PBORCA_SccGetConnectProperties ( HPBORCA hORCASession,
    LPTSTR pWorkspaceFile,
    PBORCA_SCC *pConfig );

```

Table 2.60:

Argument	Description
hORCASession	Handle to previously established ORCA session
pWorkspaceFile	Fully qualified or relative file name of the PowerBuilder workspace file (PBW)
*pConfig	Pointer to a preallocated structure typically initialized to zeros

Return value

INT. Typical return codes are:

Table 2.61:

Return code	Description
0 PBORCA_OK	Operation successful
-3 PBORCA_OBJNOTFOUND	Could not find workspace file

Usage

This method simplifies the SCC connection process. Property values returned from the workspace you include as an argument in the `PBORCA_SccGetConnectProperties` call are stored in a preallocated structure, `PBORCA_SCC`. These properties allow a successful connection to a given SCC provider and project, but you can override any of these properties.

The `PBORCA_SCC` structure is defined as follows:

```
typedef struct pborca_scc {
    HWND hWnd;
    TCHAR szProviderName [PBORCA_SCC_NAME_LEN + 1];
    LONG *plCapabilities;
    TCHAR szUserID [PBORCA_SCC_USER_LEN + 1];
    TCHAR szProject [PBORCA_SCC_PATH_LEN + 1];
    TCHAR szLocalProjPath [PBORCA_SCC_PATH_LEN + 1];
    TCHAR szAuxPath [PBORCA_SCC_PATH_LEN + 1];
    TCHAR szLogFile [PBORCA_SCC_PATH_LEN + 1];
    LPTEXTOUTPROC pMsgHandler;
    LONG *pCommentLen;
    LONG lAppend;
    LPVOID pCommBlk;
} PBORCA_SCC;
```

The variables in the `PBORCA_SCC` structure are described in the following table:

Table 2.62:

Member	Description
<code>hWnd</code>	Parent window handle whose value is typically NULL.
<code>szProviderName</code>	Name of the SCC provider.
<code>*plCapabilities</code>	Pointer to value returned by <code>PBORCA_SccConnect</code> . Used internally to determine what features the SCC provider supports.
<code>szUserID</code>	User ID for the source control project.
<code>szProject</code>	Name of the source control project.
<code>szLocalProjPath</code>	Local root directory for the project.
<code>szAuxPath</code>	The Auxiliary Project Path has different meaning for every SCC vendor. It can contain any string that the SCC provider wants to associate with the project. <code>PBORCA_SccGetConnectProperties</code> returns this value to enable a silent connection (without opening a dialog box from the SCC provider).

Member	Description
szLogFile	Name of the log file for the SCC connection.
pMsgHandler	Callback function for SCC messages.
*pCommentLen	Pointer to value returned by PBORCA_SccConnect. Length of comments accepted by the SCC provider.
lAppend	Determines whether to append to (lAppend=1) or overwrite (lAppend=0) the SCC log file.
pCommBlk	Reserved for internal use.

The property values added to the PBORCA_SCC structure after calling the PBORCA_SccGetConnectProperties function are szProviderName, szUserID, szProject, szLocalProjPath, szAuxPath, szLogFile, and lAppend. If you manually add these values to the PBORCA_SCC structure, you do not need to call the PBORCA_SccGetConnectProperties to connect to source control.

See also

[PBORCA_SccConnect](#)

[PBORCA_SccSetTarget](#)

2.31 PBORCA_SccGetLatestVersion

Description

Retrieves the latest version of files from the SCC provider.

Syntax

```
INT PBORCA_SccGetLatestVer ( HPBORCA hORCASession,
    Long nFiles,
    LPTSTR *ppFileNames );
```

Table 2.63:

Argument	Description
hORCASession	Handle to previously established ORCA session
nFiles	Number of files to be retrieved
*ppFileNames	Names of files to be retrieved

Return value

INT. Typical return codes are:

Table 2.64:

Return code	Description
0 PBORCA_OK	Operation successful

Return code	Description
-22 PBORCA_SCCFAILURE	Operation failure

Usage

Call this method to retrieve files from source control. Typically, these are objects that exist outside of a PowerBuilder library but nevertheless belong to an application. Examples include BMP, JPG, ICO, DOC, HLP, HTM, JSP, and PBR files.

Examples

The following example:

```
LPTSTR  pOtherFiles[3];
pOtherFiles[0] = _TEXT("c:\\qadb\\qadbtest\\qadbtest.hlp");
pOtherFiles[1] = _TEXT("c:\\qadb\\datatypes\\datatypes.pbr");
pOtherFiles[2] = _TEXT("c:\\qadb\\qadbtest.bmp");

lpORCA_Info->lReturnCode = PBORCA_SccGetLatestVer
    (lpORCA_Info->hORCASession, 3, pOtherFiles);
```

See also

[PBORCA_SccConnect](#)

[PBORCA_SccSetTarget](#)

2.32 PBORCA_SccRefreshTarget

Description

Calls SccGetLatestVersion to refresh the source for each of the objects in the target libraries.

Syntax

```
INT PBORCA_SccRefreshTarget ( HPBORCA hORCASession, PBORCA_REBLD_TYPE eRebldType );
```

Table 2.65:

Argument	Description
hORCASession	Handle to previously established ORCA session
eRebldType	Allows you to specify how the application is rebuilt (see Usage section below)

Return value

INT.

Usage

Call this method to get the latest version of objects in target libraries from source control. The refresh operation also causes the objects to be imported and compiled in their respective PowerBuilder libraries.

Objects in target libraries that you name in a PBORCA_SccExcludeLibraryList call are not included in the refresh operation.

The `PBORCA_REBLD_TYPE` argument determines how the application is rebuilt when you call `PBORCA_SccRefreshTarget`:

Table 2.66:

PBORCA_REBLD_TYPE	Description
<code>PBORCA_FULL_REBUILD</code>	Performs a full rebuild of the application
<code>PBORCA_INCREMENTAL_REBUILD</code>	Performs an incremental rebuild of the application
<code>PBORCA_MIGRATE</code>	Upgrades the application and performs full rebuild

See also

[PBORCA_SccClose](#)

[PBORCA_SccConnect](#)

[PBORCA_SccExcludeLibraryList](#)

[PBORCA_SccSetTarget](#)

2.33 PBORCA_SccResetRevisionNumber

Description

Call this function to reset the revision number for an object. This function is useful only in applications using SCC providers that implement the `SccQueryInfoEx` extension to the SCC API.

Syntax

```
INT PBORCA_SccResetRevisionNumber ( HPBORCA hORCASession,
    LPTSTR lpszLibraryName,
    LPTSTR lpszEntryName,
    PBORCA_TYPE otEntryType,
    LPTSTR lpszRevisionNum );
```

Table 2.67:

Argument	Description
<code>hORCASession</code>	Handle to previously established ORCA session.
<code>lpszLibraryName</code>	Absolute or relative path specification for the PBL file containing the object for which you want to reset the revision number.
<code>lpszEntryName</code>	Pointer to a string whose value is the name of the object without its <code>.sr?</code> extension.
<code>otEntryType</code>	A value of the <code>PBORCA_TYPE</code> enumerated data type specifying the object type of the entry being imported. Values are: <code>PBORCA_APPLICATION</code>

Argument	Description
	PBORCA_BINARY PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW
lpszRevisionNum	A string value or NULL. NULL causes the current revision number in the PBL to be deleted.

Return value

INT. Typical return codes are:

Table 2.68:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list (if lpszLibraryName or lpszEntryName is null)
-7 PBORCA_LIBIOERROR	Unable to open PBL for read/write access

Usage

You can call this function whether or not you are connected to source control. The `PBORCA_SccResetRevisionNumber` function changes the object revision number that is stored as metadata in the PowerBuilder library that you assign in the `lpszLibraryName` argument. The revision number is changed in the object source on the desktop machine, not in the source control repository. The library where the object resides does not have to be in the current library list.

Typically you would call `PBORCA_SccResetRevisionNumber` if your ORCA program externally modifies the object source in the PBL and one of the following is also true:

- The ORCA program has imported a specific revision of an object into the PBL through a `PBORCA_CompiledEntryImport` call. If the ORCA program knows the exact revision number that was imported, that revision number should be specified in the `lpszRevisionNum` argument. If the exact revision number is unknown, the ORCA program should still call `PBORCA_SccResetRevisionNum` and set `lpszRevisionNum` to NULL.

- The ORCA program is externally performing the equivalent of an SCC check-in by exporting existing object source from the PBL through a `PBORCA_LibraryEntryExport` call and checking the object source into the SCC repository itself. To complete the job, the ORCA program must obtain the new revision number from the SCC repository and call `PBORCA_SccResetRevisionNumber`. After you do this, the object source residing in the PBL is associated with the correct revision number in the SCC repository.

See also

[PBORCA CompileEntryImport](#)

[PBORCA LibraryEntryExport](#)

2.34 PBORCA_SccSetTarget

Description

Retrieves the target file from source control, passes the application object name to ORCA, and sets the ORCA session library list.

Syntax

```
INT PBORCA_SccSetTarget ( HPBORCA hORCASession,
    LPTSTR pTargetFile,
    LONG lFlags,
    PBORCA_SETTGTPROC pSetTgtProc,
    LPVOID pUserData );
```

Table 2.69:

Argument	Description
hORCASession	Handle to previously established ORCA session
pTargetFile	Target file name
lFlags	Allows you to control the behavior of the target operation (see Usage section below)
pSetTgtProc	Pointer to the user-defined callback function
pUserData	Pointer to a preallocated data buffer

Return value

INT.

Usage

This method takes the place of `PBORCA_SetLibraryList` and `PBORCA_SetCurrentAppl` in a traditional ORCA application.

In addition to retrieving the target file from source control and setting the application object and library list, `PBORCA_SccSetTarget` calls a user-defined callback function one time for each library in the library list. This lets you know which libraries will be refreshed by default and gives you an opportunity to call `PBORCA_SccExcludeLibraryList` if you think that specific shared libraries have already been refreshed by a previous task.

You assign the IFlags argument to set the refresh behavior on target libraries you retrieve from source control:

Table 2.70:

Flag	Description
PBORCA_SCC_OUTOFDATE	Performs comparisons to determine if objects residing in the PBL are out of sync. When used with PBORCA_SCC_IMPORTONLY, only objects that differ from the source residing on the local project path are refreshed. When PBORCA_SCC_IMPORTONLY is not set, only objects that are out of date with the SCC repository are refreshed. PBORCA_SCC_OUTOFDATE and PBORCA_SCC_REFRESH_ALL are mutually exclusive.
PBORCA_SCC_REFRESH_ALL	Target libraries are completely refreshed. When used with PBORCA_SCC_IMPORTONLY, source code is imported directly from the local project path. When PBORCA_SCC_IMPORTONLY is not set, then the latest version of all objects is first obtained from the SCC provider and then imported to the target libraries.
PBORCA_SCC_IMPORTONLY	Indicates that all the necessary objects to rebuild the target application already exist on the local project path. Set this flag if you have previously refreshed the local path using the SCC vendor's administration tool. PBORCA_SCC_IMPORTONLY is required if you previously called PBORCA_SccConnectOffline during this ORCA session. PBORCA_SCC_IMPORTONLY is particularly useful to rebuild a target from a specific SCC version label or promotion group.
PBORCA_SCC_EXCLUDE_CHECKOUT	Provides a mechanism to refresh local targets through a batch job that does not require user intervention. Prevents objects that are currently checked out from being overwritten. When used along with PBORCA_SccConnect, the checkout status is obtained directly from the SCC provider. When used

Flag	Description
	with <code>PBORCA_SccConnectOffline</code> , the checkout status is obtained from the <code>workspace_name.PBC</code> file. For offline processing, the workspace name is obtained from a previous call to <code>PBORCA_SccGetConnectProperties</code> .

If target libraries and directories do not exist in the local project path specified by `PBORCA_SccConnect`, then these directories and PBL files are created dynamically by the `PBORCA_SccSetTarget` call.

`SccSetTarget` does an implicit `PBORCA_SessionSetLibraryList` and `PBORCA_SessionSetCurrentAppl`. After you call `PBORCA_SccSetTarget` (and presumably `PBORCA_SccRefreshTarget`), you can do other work that requires a current application and an initialized library list, such as creating PBDs and EXEs. This is more efficient than calling `PBORCA_SccClose`, then reinitializing the library list and current application to create the PBDs and EXEs.

See also

[PBORCA_SccConnect](#)

[PBORCA_SccConnectOffline](#)

[PBORCA_SccGetConnectProperties](#)

[PBORCA_SccRefreshTarget](#)

2.35 PBORCA_SessionClose

Description

Terminates an ORCA session.

Syntax

```
void PBORCA_SessionClose ( HPBORCA hORCASession );
```

Table 2.71:

Argument	Description
<code>hORCASession</code>	Handle to previously established ORCA session

Return value

None.

Usage

`PBORCA_SessionClose` frees any currently allocated resources related to the ORCA session. If you do not close the session, memory allocated by PowerBuilder DLLs is not freed, resulting in a memory leak. Failing to close the session does not affect data (since an ORCA session has no connection to anything).

Examples

This example closes the ORCA session:

```
PBORCA_SessionClose(lpORCA_Info->hORCASession);
lpORCA_Info->hORCASession = 0;
```

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA_SessionOpen](#)

2.36 PBORCA_SessionGetError

Description

Gets the current error for an ORCA session.

Syntax

```
void PBORCA_SessionGetError ( HPBORCA hORCASession, LPTSTR lpszErrorBuffer, INT
iErrorBufferSize );
```

Table 2.72:

Argument	Description
hORCASession	Handle to previously established ORCA session.
lpszErrorBuffer	Pointer to a buffer in which ORCA will put the current error string.
iErrorBufferSize	Size of the buffer pointed to by lpszErrorBuffer. The constant PBORCA_MSGBUFFER provides a suggested buffer size of 256. It is defined in the ORCA header file PBORCA.H

Return value

None.

Usage

You can call PBORCA_SessionGetError anytime another ORCA function call results in an error. When an error occurs, functions always return some useful error code. The complete list of codes is shown in [ORCA return codes](#). However, you can get ORCA's complete error message by calling [PBORCA_SessionGetError](#).

If there is no current error, the function puts an empty string ("") into the error buffer.

Examples

This example stores the current error message in the string buffer pointed to by lpszErrorMessage. The size of the buffer was set previously and stored in dwErrorBufferLen:

```
PBORCA_SessionGetError(lpORCA_Info->hORCASession,
```



```
lpORCA_Info->lpszErrorMessage,  
(int) lpORCA_Info->dwErrorBufferLen);
```

In these examples, session information is saved in the data structure `ORCA_Info`, shown in [About the examples](#).

2.37 `PBORCA_SessionOpen`

Description

Establishes an ORCA session and returns a handle that you use for subsequent ORCA calls.

Syntax

```
HPBORCA PBORCA_SessionOpen ( void );
```

Return value

HPBORCA. Returns a handle to the ORCA session if it succeeds and returns 0 if it fails. Opening a session fails only if no memory is available.

Usage

You must open a session before making any other ORCA function calls.

There is no overhead or resource issue related to keeping an ORCA session open; therefore, once it is established, you can leave the session open as long as it is needed.

For some ORCA tasks, such as importing and querying objects or building executables, you must call `PBORCA_SessionSetLibraryList` and `PBORCA_SessionSetCurrentAppl` to provide an application context after opening the session.

Likewise, `PBORCA_SccSetTarget` provides an implicit application context for SCC operations. Do not call `PBORCA_SessionSetLibraryList` and `PBORCA_SetCurrentAppl` if you intend to call `PBORCA_SccSetTarget`.

Examples

This example opens an ORCA session:

```
lpORCA_Info->hORCASession = PBORCA_SessionOpen();  
if (lpORCA_Info->hORCASession = NULL)  
{  
lpORCA_Info->lReturnCode = 999;  
_tcscpy(lpORCA_Info->lpszErrorMessage,  
_TEXT("Open session failed"));  
}
```

See also

[PBORCA_SessionClose](#)

[PBORCA_SessionSetLibraryList](#)

[PBORCA_SessionSetCurrentAppl](#)

2.38 `PBORCA_SessionSetCurrentAppl`

Description

Establishes the current Application object for an ORCA session.

Syntax

```
INT PBORCA_SessionSetCurrentAppl ( HPBORCA hORCASession,
    LPTSTR lpszApplLibName, LPTSTR lpszApplName );
```

Table 2.73:

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszApplLibName	Pointer to a string whose value is the name of the application library
lpszApplName	Pointer to a string whose value is the name of the Application object

Return value

INT. Typical return codes are:

Table 2.74:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-2 PBORCA_DUOPERATION	Current application is already set
-3 PBORCA_OBJNOTFOUND	Referenced library does not exist
-4 PBORCA_BADLIBRARY	Bad library name
-5 PBORCA_LIBLISTNOTSET	Library list not set
-6 PBORCA_LIBNOTINLIST	Referenced library not in library list

Usage

You must set the library list before setting the current application.

You must call `PBORCA_SessionSetLibraryList` and then `PBORCA_SessionSetCurrentAppl` before calling any ORCA function that compiles or queries objects. The library name should include the full path for the file wherever possible.

Changing the application

You can set the library list and current application only once in a session. If you need to change the current application after it has been set, close the session and open a new session.

New applications

To create a new application starting with an empty library, set the pointers to the application library name and the application name to `NULL`. ORCA will set up an internal default application.

For more information about creating a new application, see [Bootstrapping a new application](#).

Examples

This example sets the current Application object to the object named demo in the library MASTER.PBL:

```
LPTSTR pszLibraryName;
LPTSTR pszApplName;
// specify library name
pszLibraryName = _TEXT("c:\\app\\master.pbl");
// specify application name
pszApplName = _TEXT("demo");
// set the current Application object
lpORCA->lReturnCode = PBORCA_SessionSetCurrentAppl(
    lpORCA_Info->hORCASession,
    pszLibraryName, pszApplName);
```

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA_SessionSetLibraryList](#)

2.39 PBORCA_SessionSetLibraryList

Description

Establishes the list of libraries for an ORCA session. ORCA searches the libraries in the list to resolve object references.

Syntax

```
INT PBORCA_SessionSetLibraryList ( HPBORCA hORCASession,
    LPTSTR *pLibNames,
    INT iNumberOfLibs );
```

Table 2.75:

Argument	Description
hORCASession	Handle to previously established ORCA session.
*pLibNames	Pointer to an array of pointers to strings. The values of the strings are file names of libraries. Include the full path for each library where possible.
iNumberOfLibs	Number of library name pointers in the array pLibNames points to.

Return value

INT. Typical return codes are:

Table 2.76:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list

Return code	Description
-4 PBORCA_BADLIBRARY	Bad library name or a library on the list does not exist

Usage

You must call `PBORCA_SessionSetLibraryList` and `PBORCA_SessionSetCurrentAppl` before calling any ORCA function that compiles or queries objects.

Library names should be fully qualified wherever possible.

Changing the library list

You can set the current application and library list only once in a session. If you need to change either the library list or current application after it has been set, close the session and open a new session.

How ORCA uses the library list

ORCA uses the search path to find referenced objects when you regenerate or query objects during an ORCA session. Just like PowerBuilder, ORCA looks through the libraries in the order in which they are specified in the library search path until it finds a referenced object.

Functions that don't need a library list

You can call the following library management functions and source control functions without setting the library list:

`PBORCA_LibraryCommentModify`

`PBORCA_LibraryCreate`

`PBORCA_LibraryDelete`

`PBORCA_LibraryDirectory`

`PBORCA_LibraryEntryCopy`

`PBORCA_LibraryEntryDelete`

`PBORCA_LibraryEntryExport`

`PBORCA_LibraryEntryInformation`

`PBORCA_LibraryEntryMove`

Examples

This example builds an array of library file names for PocketBuilder and sets the session's library list:

```
LPTSTR lpLibraryNames[4];
// specify the library names
lpLibraryNames[0] =
    _TEXT("c:\\qadb\\qadbtest\\qadbtest.pk1");
lpLibraryNames[1] =
    _TEXT("c:\\qadb\\shared_obj\\shared_obj.pk1");
lpLibraryNames[2] =
    _TEXT("c:\\qadb\\chgreqs\\chgreqs.pk1");
lpLibraryNames[3] =
    _TEXT("c:\\qadb\\datatypes\\datatypes.pk1");
lpORCA_Info->lReturnCode = PBORCA_SessionSetLibraryList(
```

```
lpORCA_Info->hORCASession, lpLibraryNames, 4);
```

In these examples, session information is saved in the data structure ORCA_Info, shown in [About the examples](#).

See also

[PBORCA_SessionSetCurrentAppl](#)

2.40 PBORCA_SetDebug

Description

Allows you to reset the bDebug property for the ORCA session after a PBORCA_ConfigureSession has been issued. Methods that invoke the PowerScript compiler use the bDebug setting to evaluate conditional compilation logic.

Syntax

```
INT PBORCA_SetDebug ( HPBORCA hORCASession,
    BOOL bDebug );
```

Table 2.77:

Argument	Description
hORCASession	Handle to previously established ORCA session
bDebug	Setting for the DEBUG conditional compiler directive

Return value

INT. Typical return codes are:

Table 2.78:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	hORCASession is not valid

Usage

Allows the bDebug value to be reset during an ORCA session. In typical ORCA applications, bDebug is set in the PBORCA_ConfigureSession method that is called immediately after opening an ORCA session. If you set bDebug with the PBORCA_ConfigureSession method, there is typically no need to call PBORCA_SetDebug later on. If you do not call PBORCA_ConfigureSession or PBORCA_SetDebug, the bDebug value defaults to TRUE.

The PowerScript compiler uses the bDebug value to determine whether to enable or disable DEBUG conditional compilation directives when building or regenerating objects in standard PowerBuilder targets. The bDebug value is not used in Windows Forms targets, since the PBORCA_DeployWinFormProject method uses a setting in the Project object of these targets to determine whether to enable or disable the DEBUG directive.

The following ORCA methods invoke the PowerScript compiler:

- [PBORCA_ApplicationRebuild](#)
- [PBORCA_CompileEntryImport](#)
- [PBORCA_CompileEntryImportList](#)
- [PBORCA_CompileEntryRegenerate](#)
- [PBORCA_SccGetLatestVersion](#)
- [PBORCA_SccRefreshTarget](#)

Although `PBORCA_LibraryEntryCopy` and `PBORCA_LibraryEntryMove` can add or replace objects in a PBL, they do not invoke the PowerScript compiler and do not change the compiled PCODE for the added or replaced objects. If you use these methods to copy or move objects to a destination PBL, the DEBUG conditional compilation setting for these objects should be considered as unknown.

If you are uncertain as to whether the PCODE component of an object matches the current `bDebug` setting, you can call `PBORCA_CompileEntryRegenerate` to regenerate it with the current setting.

`PBORCA_SetDebug` can be called any time after `PBORCA_SessionOpen`. The `PBORCA_SetDebug` method does not mark an object as needing recompilation. Although the `PBORCA_ApplicationRebuild` method invokes the PowerScript compiler, if you use it with the `PBORCA_INCREMENTAL_REBUILD` option, it will not rebuild an object if the only change would be in the status of its DEBUG directive. Therefore, you should not use the `PBORCA_INCREMENTAL_REBUILD` option for targets that contain DEBUG conditional compilation logic.

Similarly, you should not use the `PBORCA_INCREMENTAL_REBUILD` option with the `PBORCA_SccRefreshTarget` method. If the only difference between the original object and a refreshed object is in its DEBUG conditional compilation status, the object will not refresh when this option is used.

Examples

This example is used by the OrcaScript interpreter to implement the set debug command in OrcaScript:

```
Int ParserActions::setDebug(HPBORCA hORCA, Bool bDebug)
{
    int orcaResult = PBORCA_OK;
    orcaResult = PBORCA_SetDebug( hORCA, bDebug);
    if( orcaResult != PBORCA_OK )
        orcaError(PBTEXT("set debug "), orcaResult );
    return (orcaResult == PBORCA_OK);
}
```

See also

[PBORCA_ConfigureSession](#)

2.41 PBORCA_SetExeInfo

Description

Sets the property fields with user-specified values prior to calling `PBORCA_ExecutableCreate`.

Syntax

```
INT PBORCA_SetExeInfo ( HPBORCA hORCASession, PBORCA_EXEINFO *pExeInfo );
```

Table 2.79:

Argument	Description
hORCASession	Handle to previously established ORCA session
*pExeInfo	Pointer to a structure containing executable properties

Return value

INT. Typical return codes are:

Table 2.80:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list (when pExeInfo or hORCASession is NULL)

Usage

Call this function prior to calling `PBORCA_ExecutableCreate`.

For PowerBuilder, `PBORCA_SetExeInfo` also sets properties for dynamic libraries if machine code compilation is requested.

The `PBORCA_EXEINFO` structure is defined as follows:

```
typedef struct pborca_exeinfo
{
LPTSTR    lpszCompanyName;
LPTSTR    lpszProductName;
LPTSTR    lpszDescription;
LPTSTR    lpszCopyright;
LPTSTR    lpszFileVersion;
LPTSTR    lpszFileVersionNum;
LPTSTR    lpszProductVersion;
LPTSTR    lpszProductVersionNum;
} PBORCA_EXEINFO
```

The user must have already issued `PBORCA_SessionOpen`, `PBORCA_SessionSetCurrentAppl`, and `PBORCA_SetLibraryList` before calling `PBORCA_SetExeInfo`.

Information in the `PBORCA_EXEINFO` structure is copied to an internal ORCA control structure so that the caller can free this memory immediately upon completion of the `PBORCA_SetExeInfo` call.

The executable version information is deleted during `PBORCA_SessionClose` processing. Thus, if an ORCA program creates numerous ORCA sessions, each individual session must

call `PBORCA_SetExeInfo` and reassign all of the elements in the `PBORCA_EXEINFO` structure.

The `FileVersionNum` and `ProductVersionNum` strings must consist of four integer values representing the major version number, minor version number, fix version number, and build number, with each integer value separated by a comma. For example, "12,0,0,0001".

Examples

This example sets the executable information for a PowerBuilder application:

```
memset(&ExeInfo, 0x00, sizeof(PBORCA_EXEINFO));
ExeInfo.lpszCompanyName = _TEXT("Appeon");
ExeInfo.lpszProductName = _TEXT("PowerBuilder 2019 R3 DBAuto");
ExeInfo.lpszDescription = _TEXT("Batch Automation for QADB Test Suite");
ExeInfo.lpszCopyright = _TEXT("2011");
ExeInfo.lpszFileVersion = _TEXT("12.5.0.001");
ExeInfo.lpszFileVersionNum = _TEXT("12,5,0,001");
ExeInfo.lpszProductVersion = _TEXT("12.5.0.001");
ExeInfo.lpszProductVersionNum = _TEXT("12,5,0,001");
LpORCA_Info->lReturnCode = PBORCA_SetExeInfo(
lpORCA_Info->hORCASession, &ExeInfo );
lpORCA_Info->hORCASession, lpLibraryNames, 2);
```

See also

[PBORCA_DynamicLibraryCreate](#)

[PBORCA_ExecutableCreate](#)

3 ORCA Callback Functions and Structures

About this chapter

This chapter documents the prototypes for the callback functions used for several ORCA functions as well as the structures passed to those functions. These prototypes are declared in PBORCA.H.

3.1 Callback function for compiling objects

Description

Called for each error that occurs when objects in a library are compiled so that the errors can be stored for later display.

Functions that use this callback format are:

PBORCA_ApplicationRebuild

PBORCA_CompileEntryImport

PBORCA_CompileEntryImportList

PBORCA_CompileEntryRegenerate

Syntax

```
typedef void (CALLBACK *PBORCA_ERRPROC) ( PPBORCA_COMPERR, LPVOID );
```

Table 3.1:

Argument	Description
PPBORCA_COMPERR	Pointer to the structure PBORCA_COMPERR (described next)
LPVOID	Long pointer to user data

Return value

None.

Usage

You provide the code for the callback function. The callback function generally reads the error information passed in the PBORCA_COMPERR structure, extracts whatever is wanted, and formats it in the user data buffer pointed to by LPVOID.

The user data buffer is allocated in the calling program and can be structured any way you want. It might include a structure that counts the errors and an array or text block in which you format information about all the errors.

For information and examples of coding a callback function, see [About ORCA callback functions](#).

3.2 PBORCA_COMPERR structure

Description

Reports information about an error that occurred when you tried to import and compile objects in a library.

The following functions pass the `PBORCA_COMPERR` structure to their callback functions:

`PBORCA_CompileEntryImport`

`PBORCA_CompileEntryImportList`

`PBORCA_CompileEntryRegenerate`

Syntax

```
typedef struct pborca_comperr {
    int iLevel;
    LPTSTR lpszMessageNumber;
    LPTSTR lpszMessageText;
    UINT iColumnNumber;
    UINT iLineNumber;
} PBORCA_COMPERR, FAR *PPBORCA_COMPERR;
```

Table 3.2:

Member	Description
<code>iLevel</code>	Number identifying the severity of the error. Values are: 0 -- Context information, such as object or script name 1 -- <code>CM_INFORMATION_LEVEL</code> 2 -- <code>CM_OBSOLETE_LEVEL</code> 3 -- <code>CM_WARNING_LEVEL</code> 4 -- <code>CM_ERROR_LEVEL</code> 5 -- <code>CM_FATAL_LEVEL</code> 6 -- <code>CM_DBWARNING_LEVEL</code>
<code>lpszMessageNumber</code>	Pointer to a string whose value is the message number
<code>lpszMessageText</code>	Pointer to a string whose value is the text of the error message
<code>iColumnNumber</code>	Number of the character in the line of source code where the error occurred
<code>iLineNumber</code>	Number of the line of source code where the error occurred

Usage

A single error might trigger several calls to the callback function. The first messages report the object and script in which the error occurred. Then one or more messages report the actual error.

For example, an IF-THEN-ELSE block missing an END IF generates these messages:

Table 3.3:

Lvl	Num	Message text	Col	Line
0	null	Object: f_boolean_to_char	0	0
0	null	Function Source	0	0
4	null	(0002): Error C0031: Syntax error	0	2
4	null	(0016): Error C0031: Syntax error	0	16
4	null	(0017): Error C0031: Syntax error	0	17

3.3 Callback function for deploying components to EAServer (Obsolete)

Description

This function is obsolete because EAServer is no longer supported since PowerBuilder 2017. An obsolete feature is no longer eligible for technical support and will no longer be enhanced, although it is still available.

Called for each error that occurs when objects are deployed to EAServer so that the errors can be stored for later display.

Functions that use this callback format are:

PBORCA_BuildProject

PBORCA_BuildProjectEx

Syntax

```
typedef PSCALLBACK (void, *PPBORCA_BLDPROC) ( PBORCA_BLDERR, LPVOID );
```

Table 3.4:

Argument	Description
PPBORCA_BLDERR	Pointer to the structure PBORCA_BLDERR (described next)
LPVOID	Long pointer to user data

Return value

None.

Usage

For information and examples of coding a callback function, see [About ORCA callback functions](#).

3.4 PBORCA_BLDERR structure

Description

This function is obsolete because EAServer is no longer supported since PowerBuilder 2017.

Reports information about an error that occurred when you tried to deploy objects to EAServer.

The following functions pass the PBORCA_BLDERR structure to their callback functions:

PBORCA_BuildProject

PBORCA_BuildProjectEx

Syntax

```
typedef struct pborca_blderr { LPTSTR lpszMessageText; } PBORCA_BLDERR, FAR
*PPBORCA_BLDERR;
```

Table 3.5:

Member	Description
lpszMessageText	Pointer to a string whose value is the text of the error message

3.5 Callback function for PBORCA_LibraryDirectory

Description

Called for each entry in the library so that information about the entry can be stored for later display.

Syntax

```
typedef void (CALLBACK *PBORCA_LISTPROC) ( PPBORCA_DIRENTRY, LPVOID );
```

Table 3.6:

Argument	Description
PPBORCA_DIRENTRY	Pointer to the structure PBORCA_DIRENTRY (described next)
LPVOID	Long pointer to user data

Return value

None.

Usage

You provide the code for the callback function. The callback function generally reads the information about the library entry passed in the PBORCA_DIRENTRY structure, extracts whatever is wanted, and formats it in the user data buffer pointed to by LPVOID.

The user data buffer is allocated in the calling program and can be structured any way you want. It might include a structure that counts the entries and an array or text block in which you format information about all the entries.

For information and examples of coding a callback function, see [About ORCA callback functions](#).

3.6 PBORCA_DIRENTRY structure

Description

Reports information about an entry in a library.

The `PBORCA_LibraryDirectory` function passes the `PBORCA_DIRENTRY` structure to its callback function.

Syntax

```
typedef struct pborca_direntry {
    TCHAR szComments[PBORCA_MAXCOMMENT + 1];
    LONG lCreateTime;
    LONG lEntrySize;
    LPTSTR lpszEntryName;
    PBORCA_TYPE otEntryType;
} PBORCA_DIRENTRY, FAR *PPBORCA_DIRENTRY;
```

Table 3.7:

Member	Description
szComments	Comments stored in the library for the object
lCreateTime	The time the object was created
lEntrySize	The size of the object, including its source code and the compiled object
lpszEntryName	The name of the object for which information is being returned
otEntryType	A value of the enumerated data type <code>PBORCA_TYPE</code> specifying the data type of the object

3.7 Callback function for PBORCA_ObjectQueryHierarchy

Description

Called for each ancestor object in the hierarchy of the object being examined. In the callback function, you can save the ancestor name for later display.

Syntax

```
typedef void (CALLBACK *PBORCA_HIERPROC)
    ( PPBORCA_HIERARCHY, LPVOID );
```

Table 3.8:

Argument	Description
PPBORCA_HIERARCHY	Pointer to the PBORCA_HIERARCHY structure (described next)
LPVOID	Long pointer to user data

Return value

None.

Usage

You provide the code for the callback function. The callback function generally reads the ancestor name passed in the `PBORCA_HIERARCHY` structure and saves it in the user data buffer pointed to by `LPVOID`.

The user data buffer is allocated in the calling program and can be structured any way you want. It might include a structure that counts the number of ancestors and an array or text block in which you store the names.

For information and examples of coding a callback function, see [About ORCA callback functions](#).

3.8 PBORCA_HIERARCHY structure

Description

Reports the name of an ancestor object for the object being queried.

The `PBORCA_ObjectQueryHierarchy` function passes the `PBORCA_HIERARCHY` structure to its callback function.

Syntax

```
typedef struct pborca_hierarchy {
    LPTSTR lpszAncestorName;
} PBORCA_HIERARCHY, FAR *PPBORCA_HIERARCHY;
```

Table 3.9:

Member	Description
<code>lpszAncestorName</code>	Pointer to name of ancestor object

3.9 Callback function for PBORCA_ObjectQueryReference

Description

Called for each referenced object in the object being examined. In the callback function, you can save the name of the referenced object for later display.

Syntax

```
typedef void (CALLBACK *PBORCA_REFPROC)
( PPBORCA_REFERENCE, LPVOID );
```

Table 3.10:

Argument	Description
<code>PPBORCA_REFERENCE</code>	Pointer to the PBORCA_REFERENCE structure (described next)
<code>LPVOID</code>	Long pointer to user data

Return value

None.

Usage

You provide the code for the callback function. The callback function generally reads the name of the referenced object passed in the `PBORCA_REFERENCE` structure and saves it in the user data buffer pointed to by `LPVOID`.

The user data buffer is allocated in the calling program and can be structured any way you want. It might include a structure that counts the number of referenced objects and an array or text block in which you store the names.

For information and examples of coding a callback function, see [About ORCA callback functions](#).

3.10 PBORCA_REFERENCE structure

Description

Reports the name of an object that the object being queried refers to.

The `PBORCA_ObjectQueryReference` function passes the `PBORCA_REFERENCE` structure to its callback function.

Syntax

```
typedef struct pborca_reference {
    LPTSTR lpszLibraryName;
    LPTSTR lpszEntryName;
    PBORCA_TYPE otEntryType;
} PBORCA_REFERENCE, FAR *PPBORCA_REFERENCE;
```

Table 3.11:

Member	Description
<code>lpszLibraryName</code>	Pointer to a string whose value is the file name of the library containing the referenced object
<code>lpszEntryName</code>	Pointer to a string whose value is the name of the referenced object
<code>otEntryType</code>	A value of the enumerated data type <code>PBORCA_TYPE</code> specifying the type of the referenced object

3.11 Callback function for `PBORCA_ExecutableCreate`

Description

Called for each link error that occurs while you are building an executable.

Syntax

```
typedef void (CALLBACK *PBORCA_LNKPROC)
( PPBORCA_LINKERR, LPVOID );
```

Table 3.12:

Argument	Description
<code>PPBORCA_LINKERR</code>	Pointer to the PBORCA_LINKERR structure (described next)

Argument	Description
LPVOID	Long pointer to user data

Return value

None.

Usage

You provide the code for the callback function. The callback function generally reads the error information passed in the `PBORCA_LINKERR` structure and formats the message text in the user data buffer pointed to by `LPVOID`.

The user data buffer is allocated in the calling program and can be structured any way you want. It might include a structure that counts the errors and an array or text block in which you format the message text.

For information and examples of coding a callback function, see [About ORCA callback functions](#).

3.12 PBORCA_LINKERR structure

Description

Reports the message text for a link error that has occurred when you build an executable.

The `PBORCA_ExecutableCreate` function passes the `PBORCA_LINKERR` structure to its callback function.

Syntax

```
typedef struct pborca_linkerr {
    LPTSTR lpszMessageText;
} PBORCA_LINKERR, FAR *PPBORCA_LINKERR;
```

Table 3.13:

Member	Description
<code>lpszMessageText</code>	Pointer to the text of the error message

3.13 Callback function for PBORCA_SccSetTarget

Description

Called once for each library in the target library list.

Syntax

```
typedef PBCALLBACK (void, *PBORCA_SETTGTPROC)
( PPBORCA_SETTARGET, LPVOID );
```

Table 3.14:

Argument	Description
<code>PPBORCA_SETTARGET</code>	Pointer to the PBORCA_SCCSETTARGET structure
<code>LPVOID</code>	Long pointer to user data

Return value

None.

Usage

This callback function allows you to know which libraries are going to be refreshed by default and gives you the opportunity to call `PBORCA_SccExcludeLibraryList` when you are certain that specific shared libraries have already been refreshed by a previous task.

3.14 PBORCA_SCCSETTARGET structure

Description

Reports the fully qualified name of a library in the target library list.

Syntax

```
typedef struct pborca_sccsettarget {  
    LPTSTR lpszLibraryName;  
} PBORCA_SETTARGET, FAR *PPBORCA_SETTARGET;
```

Table 3.15:

Member	Description
<code>lpszLibraryName</code>	Pointer to the name of a library in the target library list