ORCA Guide

PowerBuilder

Version 7

March 1999

About This Book

Subject

This book describes the Powersoft OpenLibrary API (ORCA), which is part of the Enterprise and Professional editions of PowerBuilder.

This book provides the information you need to develop ORCA programs. You will find information about:

- Installing ORCA software
- The parallels between ORCA functions and what a PowerBuilder developer can do in the Library painter
- Writing an ORCA program
- Reference material for ORCA functions and callback functions

Audience

This book is for tool vendors This book is for Powersoft code partners and other tool vendors who develop companion products and tools that manipulate and manage objects in PowerBuilder libraries for use with PowerBuilder.

Constraints
ORCA users need to be aware of the constraints involved in using ORCA. They are described in "Who can develop programs that call ORCA?" on page 7.

This book is not for PowerBuilder application developers ORCA is not designed for use in building PowerBuilder applications. Nor is this book needed for running programs that use ORCA (those programs should come with their own documentation).

CHAPTER 1 Using ORCA

About this chapter

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

It explains the correspondence between tasks a PowerBuilder developer can do 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.

Contents

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 Powersoft 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 may:

- 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:

- CASE tools
- Class libraries
- Documentation tools
- Application management tools
- Utilities (such as searching for text and replacing it throughout a library or displaying a tree view of objects in a library)
- Interfaces for source control systems that PowerBuilder doesn’t support directly

What can ORCA do?

ORCA lets your application do programmatically the same library and object management tasks that a developer does 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
- Check an object in or out of a library and look at its status
- 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)

Who can develop programs that call ORCA?

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

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 more than one program 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.

Installing ORCA

ORCA is available to code partners and other tool vendors who develop companion products and tools that manipulate and manage objects in PowerBuilder libraries for use with PowerBuilder on the Windows NT, Windows 95, Windows 98, and UNIX platforms.

To run ORCA programs

To run programs that use ORCA, you need the ORCA DLL (called PBORC70.DLL in PowerBuilder Version 7). When you install PowerBuilder, it is installed in the same directory with other PowerBuilder DLLs.

To develop ORCA programs

To develop C programs that use ORCA, you need several items, available from the Sybase Developers Network web site:

- C development files

  PBORCA.H
  PBORCA.LIB

- This documentation, available in PDF format

ORCA and the PowerBuilder 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 human-readable.

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.

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.

Object source code

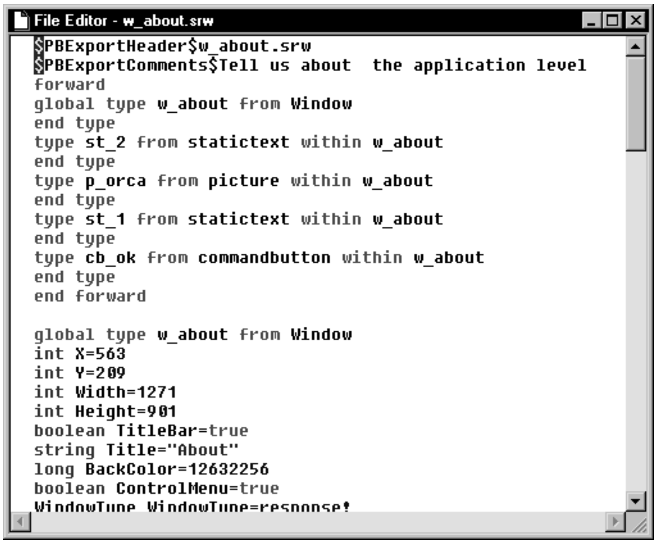
The Library painter lets you export source code, study and even modify it in any text editor, and import it back into the library. PowerBuilder compiles the imported object to check that the source code is valid. It will not import objects that fail to compile.

Source code exported to a file has two header lines before the source code. These header lines must not be included if you import the source using ORCA:

```
$PBExportHeader$w_about.srw

$PBExportComments$Tell us about the application level
```

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



```
$PBExportHeader$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
WindowType WindowType=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. The exported source is stored in a text buffer in the program. The ORCA source does not have the two header lines that the Library painter exports.

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.

About ORCA functions

ORCA functions can be divided into six functional groups:

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

Functions for managing the ORCA session

Just as you begin a session in the PowerBuilder development environment by running PowerBuilder and you 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 have a current application. You also set the library list search path if you plan to view or modify objects or create executables. Using ORCA, you have the same requirement, but the order is reversed. In ORCA, you set the library list then set the current application.

ORCA functions that don't involve compiling objects or building applications do not require a library list and current application. These are the library management functions and source control functions.

Session management

The session management functions (which all have the prefix PBORCA_) and their equivalents in the PowerBuilder development environment are:

Function (prefix PBORCA_)	Purpose	Equivalent in PowerBuilder
SessionOpen	Opens an ORCA session and returns the session handle	Starting PowerBuilder
SessionClose	Closes an ORCA session	Exiting PowerBuilder
SessionSetLibraryList	Specifies the libraries for the session	File>Library List
SessionSetCurrentAppl	Specifies the Application object for the session	File>Select Application
SessionGetError	Provides information about an error	No correspondence

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 may be called outside the context of a library list and current application.

The library management functions (which all have the prefix PBORCA_) and their equivalents in the PowerBuilder Library painter are:

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

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.

The compilation functions (which all have the prefix PBORCA_) and their equivalents in the PowerBuilder Library painter are:

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" below.

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.

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

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

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.

The functions for creating executables and libraries (which all have the prefix PBORCA_) and their equivalents in the PowerBuilder development environment are:

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

Functions for implementing source control

The source control functions provide a way to check an object in to or out of a PowerBuilder library using PowerBuilder native source control. You can also find out an object's checkout status.

These functions allow you to implement version control using version control systems that don't have a PowerBuilder interface.

You do not need to set the library list and current application before calling these functions.

The source management functions (which all have the prefix PBORCA_) and their equivalents in the PowerBuilder Library painter are:

Function (prefix PBORCA_)	Purpose	Equivalent in Library painter
CheckInEntry	Checks an object into a library	Source>Check In or Source>Clear Check Out Status
CheckOutEntry	Checks an object out of a library	Source>Check Out
ListCheckOutEntries	Reports objects that are registered or checked out	Source>View Check Out Status

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 need 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 only be used by programs written in languages that use pointers, such as C and C++.

ORCA functions that use callbacks

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

ORCA function call (prefix PBORCA_)	**Purpose of callback**
CompileEntryImport CompileEntryImportList CompileEntryRegenerate	Called once for each compile error
ExecutableCreate	Called once for each link error
LibraryDirectory	Called once for each library entry name
ObjectQueryHierarchy	Called once for every ancestor name
ObjectQueryReference	Called once for every object referenced in the entry
ListCheckOutEntries	Called once for each object that is registered or checked out in a library

How a callback works

ORCA calls a callback function like this:

1 The calling program allocates a buffer to hold data (the UserData buffer).

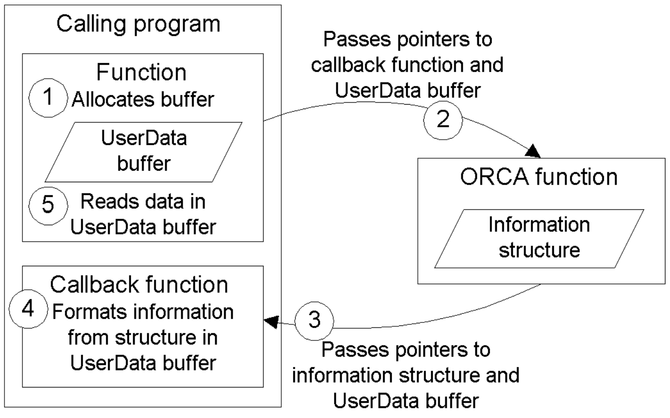
2 The calling program calls an ORCA function, passing it pointers to the callback function and the UserData buffer.

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 may 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.


Calling program
Function
1 Allocates buffer
UserData buffer
5 Reads data in UserData buffer
Callback function
4 Formats information from structure in UserData buffer
Passes pointers to callback function and UserData buffer
2
ORCA function
Information structure
3
Passes pointers to information structure and UserData buffer

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 {
    LPSTR 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;
CHAR InfoBuffer[60000];

lpUserDataBuffer = &UserDataBuffer;

lpUserDataBuffer->dwCallCount = 0;
lpUserDataBuffer->dwBufferOffset = 0;
lpUserDataBuffer->dwBufferSize = 60000;
lpUserDataBuffer->lpszBuffer = &InfoBuffer;

// Initialize all of InfoBuffer to 0

memset(lpUserDataBuffer->lpszBuffer,

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

Create Proc instance The calling program also needs to create a Proc instance of the callback function that it will pass to the ORCA function:

```
FARPROC lpCallbackProc;lpCallbackProc =
      MakeProcInstance((FARPROC)CallbackFunc, hInst);
```

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)
lpCallbackProc, lpUserDataBuffer);
```

Free Proc instance When control returns from ORCA back to the calling program, the calling program can free the Proc instance:

```
FreeProcInstance(lpCallbackProc);
```

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

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 until the buffer is full
- Store a final message when the messages no longer fit indicating that some messages, although counted, are not stored

This code implements a callback called LinkErrors for PBORCA_ExecutableCreate:

```
void WINAPI LinkErrors(PPBORCA_LINKERR lpLinkError,
      LPVOID lpUserData)
{
   PORCA_USERDATAINFO lpData;
   LPSTR 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 =
      strlen(lpLinkError->lpszMessageText) + 2;

   // Set pointer for copying message to buffer
   lpCurrentPtr = lpData->lpszBuffer
      + lpData->dwBufferOffset;

   // Check if there’s room for the message
   // plus the final message
   if ((iNeededSize + strlen("Buffer full")) >
      (lpData->dwBufferSize - lpData->dwBufferOffset))
      {
      // If almost full, copy final message
      // and set offset to end of buffer
```

```
            strcat(lpCurrentPtr, "Buffer full");
            lpData->dwBufferOffset = lpData->dwBufferSize;

            } else {
            // If not full, copy link error message,
            // CR and LF, and update offset
            strcat(lpCurrentPtr,
            lpLinkError->lpszMessageText);
            strcat(lpCurrentPtr, "\r\n");
            lpData->dwBufferOffset += iNeededSize;
            }
        return;
}
```

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.

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.

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;
}
```

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:

```
nReturnCode WINAPI SetUpSession(HPBORCA hORCASession)
{
   char szApplName[36];
   int nReturnCode;
   LPSTR lpLibraryNames[2] =
   {"c:\\pbfiles\\demo\\master.pbl",
      "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
   strcpy(szApplName, "demo");

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

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" on page 15.

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;
}
```

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.

CHAPTER 2

ORCA Functions

About this chapter

This chapter documents the ORCA functions.

Contents

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 {
    LPSTR 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
} ORCA_INFO, FAR *PORCA_INFO;
```

ORCA return codes

The header file PBORCA.H defines these return codes:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-2 PBORCA_DUPOPERATION	Duplicate operation
-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
-8 PBORCA_OBJEXISTS	Object exists
-9 PBORCA_INVALIDNAME	Invalid name
-10 PBORCA_BUFFERTOOSMALL	Buffer size is too small
-11 PBORCA_COMPERROR	Compile error
-12 PBORCA_LINKERROR	Link error
-13 PBORCA_CURRAPPLNOTSET	Current application not set
-14 PBORCA_OBJHASNOANCS	Object has no ancestors
-15 PBORCA_OBJHASNOREFS	Object has no references
-16 PBORCA_PBDCOUNTERROR	Invalid # of PBDs
-17 PBORCA_PBDCREATERROR	PBD create error
-18 PBORCA_CHECKOUTERROR	Source Management error

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 far pascal **PBORCA_ApplicationRebuild** (HPBORCA *hORCASession*,
PBORCA_REBLD_TYPE *eRebldType*,
PBORCA_ERRPROC *pCompErrProc*,
LPVOID *pUserData*);

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 PBORCA_INCREMENTAL_REBUILD PBORCA_MIGRATE
pCompErrorProc	Pointer to the PBORCA_ApplicationRebuild callback function. The callback function is called for each error that occurs as the 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_CompileEntryImport callback function 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 If you are not using a callback function, set pUserData to 0

Return value

int. Typical return codes are:

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 the objects are compiled. If two objects both 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:

```
FARPROC lpCompErrProc;
int nReturnCode;

// Get the proc address of the callback function
lpCompErrProc = MakeProcInstance(
    (FARPROC)CompileEntryErrors, hInst);

nReturnCode = PBORCA_ApplicationRebuild(
    lpORCA_Info->hORCASession,
    PBORCA_FULL_REBUILD,
    (PBORCA_ERRPROC) lpCompErrProc, lpUserData);

FreeProcInstance(lpCompErrProc);
```

For more information about setting up the data buffer for the callback, see "Content of a callback function" on page 16 and the example for PBORCA_LibraryDirectory.

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_CompileEntryRegenerate
PBORCA_CompileEntryImport
PBORCA_CompileEntryImportList

PBORCA_CheckInEntry

Description

Checks in a library entry from a work library (the destination) to a master library (the source).

Syntax

```
int far pascal PBORCA_CheckInEntry ( HPBORCA hORCASession,
            LPSTR lpszEntryName,
            LPSTR lpszSourceLibName,
            LPSTR lpszDestLibName,
            LPSTR lpszUserID,
            PBORCA_TYPE otEntryType,
            int bMoveEntry );
```

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszEntryName	Pointer to a string whose value is the name of the object being checked in
lpszSourceLibName	Pointer to a string whose value is the filename of the master library from which you checked out the object
lpszDestLibName	Pointer to a string whose value is the filename of the work library containing the object you want to check in
lpszUserID	Pointer to a string whose value is the version control user ID. The ID must be the same one used to check out the object
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being checked in. Values are: PBORCA_APPLICATION PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT

Argument	Description
bMoveEntry	An integer whose value indicates whether to simply change the check-out flags in the libraries or to move the object from the work library to the master library, deleting it from the work library. Values are: • 0 — Clear the check-out status of the object in the master and work libraries, but leave the copy of the object in the master library as is. Do not overwrite it with the copy in the work library and do not delete it from the work library • 1 — Clear the check-out status of the object in the master and work libraries and move the copy in the work library to the master library, deleting it from the work library

Return value

int. Typical return codes are:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-3 PBORCA_OBJNOTFOUND	Source or destination library not found
-4 PBORCA_BADLIBRARY	Bad library name
-18 PBORCA_CHECKOUTERROR	Entry not checked out or checked out to different user

Usage

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

Check-in only succeeds if you use the same user ID you used for check-out.

Checking in an object without moving it allows you to clear a check-out lock that is out of sync.

Examples

This example checks in a DataWindow named d_labels. The object being checked in is in the work library named in the string pointed to by lpszDestLibName. It is checked in to the master library named in the string pointed to by lpszSourceLibName. The object is deleted from the work library:

```
lReturnCode = PBORCA_CheckInEntry(
   lpORCA_Info->hORCASession,
   "d_labels",
   lpszSourceLibName, lpszDestLibName,
   "PAMM", PBORCA_DATAWINDOW, 1);
```

See also

PBORCA_CheckOutEntry
PBORCA_ListCheckOutEntries

PBORCA_CheckOutEntry

Description

Checks out a library entry from a master library (the source) to a work library (the destination).

Syntax

int far pascal **PBORCA_CheckOutEntry** (HPBORCA *hORCASession*,
LPSTR *lpszEntryName*,
LPSTR *lpszSourceLibName*,
LPSTR *lpszDestLibName*,
LPSTR *lpszUserID*,
PBORCA_TYPE *otEntryType*,
int *bMakeCopy*);

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszEntryName	Pointer to a string whose value is the name of the object being checked out
lpszSourceLibName	Pointer to a string whose value is the filename of the master library containing the object
lpszDestLibName	Pointer to a string whose value is the filename of the work library to which you want to check out the object
lpszUserID	Pointer to a string whose value is the version control user ID
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being checked out. Values are: PBORCA_APPLICATION PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT

Argument	Description
bMakeCopy	An integer whose value indicates whether to simply change the check-out flags in the libraries or to copy the object to the work library too. Values are: • 0 — Mark the object as checked out in the master and work libraries, but leave the copy of the object in the work library as is. Do not overwrite it with the copy in the master library • 1 — Mark the object as checked out in the master and work libraries and copy the object from the master library to the work library

Return value

int. Typical return codes are:

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
-18 PBORCA_CHECKOUTERROR	Entry already checked out

Usage

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

After you check out an object, PowerBuilder does not allow any modifications to the object in the master PBL where it is registered. Within PowerBuilder, only the copy in the work library can be modified.

When you want to check the object back in, you must use the same user ID you used for check-out.

Examples

This example checks out a DataWindow named d_labels from a master library (named in the string pointed to by lpszSourceLibName). The object is not copied into the work library (named in the string pointed to by lpszDestLibName). The developer can work with his or her own version of the object:

```
lReturnCode = PBORCA_CheckOutEntry(
   lpORCA_Info->hORCASession,
   "d_labels",
   lpszSourceLibName, lpszDestLibName,
   "PAMM", PBORCA_DATAWINDOW, 0);
```

See also

PBORCA_CheckInEntry
PBORCA_ListCheckOutEntries

PBORCA_CompileEntryImport

Description

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

Syntax

```
int PBORCA_CompileEntryImport ( PBORCA hORCASession,
          LPSTR lpszLibraryName,
          LPSTR lpszEntryName,
          PBORCA_TYPE otEntryType,
          LPSTR lpszComments,
          LPSTR lpszEntrySyntax,
          LONG lEntrySyntaxBuffSize,
          PBORCA_ERRPROC pCompErrorProc,
          LPVOID pUserData );
```

Argument	**Description**
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the filename 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_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT PBORCA_BINARY
lpszComments	Pointer to a string whose value is the comments you are providing for the object
lpszEntrySyntax	Pointer to a string whose value is the source code for the object to be imported
lEntrySyntaxBuffSize	Length of the string pointed to by *lpszEntrySyntax*

Argument	Description
pCompErrorProc	Pointer to the PBORCA_CompileEntryImport callback function. The callback function is called for each error that occurs as the imported object is 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_CompileEntryImport callback function 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 If you are not using a callback function, set pUserData to 0

Return value

int. Typical return codes are:

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
-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.

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. See the examples section below for sample code using PBORCA_BINARY.

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 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:

```
FARPROC lpCompErrProc;
int nReturnCode;

// Get the proc address of the callback function
lpCompErrProc = MakeProcInstance(
    (FARPROC)CompileEntryErrors, hInst);

nReturnCode = PBORCA_CompileEntryImport(
    lpORCA_Info->hORCASession,
    "c:\\app\\dwobjects.pbl",
    "d_labels", PBORCA_DATAWINDOW,
    (LPSTR) szEntrySource, 60000,
    (PBORCA_ERRPROC) lpCompErrProc, lpUserData);

FreeProcInstance(lpCompErrProc);
```

For more information about setting up the data buffer for the callback, see "Content of a callback function" on page 16 and the example for PBORCA_LibraryDirectory.

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

This example demonstrates importing and compiling a window and its embedded Word file. The window and Word file were previously exported separately (see "Examples" on page 67).

```
// Start the import from the w_word file
szMsg = "\r\nImporting library entry from file: ";
szMsg += m_szImportFileName;
szMsg += "\r\n";
SetOutputText(szMsg, 2);

char szImportBuffer[MAX_EXPORT_SIZE];
   szFile = m_szWorkingDir + "\\" + m_szImportFileName;
   CFile cImpFile(szFile, CFile::modeRead);
   cImpFile.Read(szImportBuffer,
cImpFile.GetLength());
   cImpFile.Close();

strcpy(szLibEntry, m_szLibEntryNameThree);
if ( PBORCA_CompileEntryImport(hOrca,
                szPblPath,
                szLibEntry,
                PBORCA_WINDOW,
                "Imported by ORCA from exported file",
                szImportBuffer,
                strlen(szImportBuffer),
                NULL,
                szLibEntry) )

{
   PBORCA_SessionGetError(hOrca, szBuffer, 256);
   szMsg = "ORCA Error Message: ";
   szMsg += szBuffer;
   szMsg += "\r\nImport Failed.";
   SetOutputText(szMsg,2);
}
else
   SetOutputText("Import sucessful.\r\n", 1);

// Start the import from the w_word.bin file
szMsg = "\r\nImporting library entry from file: ";
szMsg += m_szImportBinFileName;
szMsg += "\r\n";
SetOutputText(szMsg, 2);

szFile = m_szWorkingDir + "\\" + m_szImportBinFileName;
   cImpFile.Open(szFile, CFile::modeRead);
```

```
    cImpFile.Read(szImportBuffer,
cImpFile.GetLength());
cImpFile.Close();

strcpy(szLibEntry, m_szLibEntryNameThree);
if ( PBORCA_CompileEntryImport(hOrca,
                 szPblPath,
                 szLibEntry,
                 PBORCA_BINARY,
                 "Imported by ORCA from exported file",
                 szImportBuffer,
                 strlen(szImportBuffer),
                 NULL,
                 szLibEntry) )

{
    PBORCA_SessionGetError(hOrca, szBuffer, 256);
    szMsg = "ORCA Error Message: ";
    szMsg += szBuffer;
    szMsg += "\r\nImport Failed.";
    SetOutputText(szMsg,2);
}
else
    SetOutputText("Import sucessful.\r\n", 1);
```

See also

PBORCA_LibraryEntryExport
PBORCA_CompileEntryImportList
PBORCA_CompileEntryRegenerate
PBORCA_ApplicationRebuild

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,
          LPSTR far *pLibraryNames,
          LPSTR far *pEntryNames,
          PBORCA_TYPE far *otEntryTypes,
          LPSTR far *pComments,
          LPSTR far *pEntrySyntaxBuffers,
          LONG far *pEntrySyntaxBuffSizes,
          int iNumberOfEntries,
          PBORCA_ERRPROC pCompErrorProc,
          LPVOID pUserData );
```

Argument	**Description**
hORCASession	Handle to previously established ORCA session
**pLibraryNames*	Pointer to an array of strings whose values are the filenames 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 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*

Argument	Description
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_CompileEntryImportList 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_CompileEntryImportList callback function The user data typically includes the buffer or a pointer to the buffer in which the callback 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:

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 descendent 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" on page 36.

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 be 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 don't 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_CompileEntryImportList 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:

```
LPSTR lpLibraryNames[3];
LPSTR lpObjectNames[3];
PBORCA_TYPE ObjectTypes[3];
LPSTR lpObjComments[3];
LPSTR lpSourceBuffers[3];
long BuffSizes[3];

FARPROC lpCompErrProc;
int nReturnCode;

// specify the library names
lpLibraryNames[0] =
   "c:\\sybase\\pb7\\demo\\windows.pbl";
lpLibraryNames[1] =
   "c:\\sybase\\pb7\\demo\\windows.pbl";
lpLibraryNames[2] =
   "c:\\sybase\\pb7\\demo\\menus.pbl";

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

// set up object type array
ObjectTypes[0] = PBORCA_WINDOW;
```

```
ObjectTypes[1] = PBORCA_WINDOW;
ObjectTypes[2] = PBORCA_MENU;

// specify object comments
lpObjComments[0] = "Ancestor window";
lpObjComments[1] = "Descendent window";
lpObjComments[2] = "Action menu";

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

// Set up source code lengths array
BuffSizes[0] = strlen(szWindow1);
BuffSizes[1] = strlen(szWindow2);
BuffSizes[2] = strlen(szMenu1);

//Get the proc address of the callback function
lpCompErrProc = MakeProcInstance(
    (FARPROC)CompileEntryErrors, hInst);

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

FreeProcInstance(lpCompErrProc);
```

For more information about setting up the data buffer for the callback, see "Content of a callback function" on page 16 and the example for PBORCA_LibraryDirectory.

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_LibraryEntryExport
PBORCA_CompileEntryImport
PBORCA_CompileEntryRegenerate
PBORCA_ApplicationRebuild

PBORCA_CompileEntryRegenerate

Description

Compiles an object in a PowerBuilder library.

Syntax

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

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the filename 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 function. The callback function is called for each error that occurs as the object is 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

Argument	Description
pUserData	Pointer to user data to be passed to the PBORCA_CompileEntryRegenerate callback function 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 If you are not using a callback function, set pUserData to 0

Return value

int. Typical return codes are:

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
-11 PBORCA_COMPERROR	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 regenerating, 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:

```
FARPROC lpCompErrProc;
int nReturnCode;
```

```
//Get the proc address of the callback function
lpCompErrProc = MakeProcInstance(
   (FARPROC)CompileEntryErrors, hInst);

nReturnCode = PBORCA_CompileEntryRegenerate(
   lpORCA_Info->hORCASession,
   "c:\\app\\dwobjects.pbl",
   "d_labels", PBORCA_DATAWINDOW,
   (PBORCA_ERRPROC) lpCompErrProc, lpUserData );

FreeProcInstance(lpCompErrProc);
```

For more information about setting up the data buffer for the callback, see "Content of a callback function" on page 16 and the example for PBORCA_LibraryDirectory.

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_LibraryEntryExport
PBORCA_CompileEntryImport
PBORCA_CompileEntryImportList
PBORCA_ApplicationRebuild

PBORCA_DynamicLibraryCreate

Description

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

Syntax

```
int far pascal PBORCA_DynamicLibraryCreate (
        HPBORCA hORCASession,
        LPSTR lpszLibraryName,
        LPSTR lpszPBRName,
        LONG lFlags );
```

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the filename 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

Return value

int. The typical return codes are:

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 filename 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

Examples

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

```
LPSTR pszLibFile;
LPSTR pszResourceFile;
long lBuildOptions;
int rtn;

// copy filenames
pszLibFile = "c:\\app\\process.pbl";
pszResourceFile = "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 );
```

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_ExecutableCreate

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 far pascal **PBORCA_ExecutableCreate** (HPBORCA *hORCASession*,
    LPSTR *lpszExeName*,
    LPSTR *lpszIconName*,
    LPSTR *lpszPBRName*,
    PBORCA_LNKPROC *pLinkErrProc*,
    LPVOID *pUserData*,
    int FAR **iPBDFlags*,
    int *iNumberOfPBDFlags*,
    LONG *lFlags*);

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 PBORCA_ExecutableCreate callback function. The callback function is called for each link error that occurs The information ORCA passes to the callback function is the message text, stored in a structure of type PBORCA_LINKERR If you don't want to use a callback function, set *pLinkErrProc* to 0
pUserData	Pointer to user data to be passed to the PBORCA_ExecutableCreate 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 If you are not using a callback function, set pUserData to 0

Argument	Description
iPBDFlags	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: • 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
iNumberOfPBDFlags	The number of elements in the array *iPBDFlags*, which should be the same as the number of libraries on ORCA's library list
lFlags	A long value whose value indicates which code generation options to apply when building the executable Setting *lFlags* to 0 generates a native Pcode executable. Additional settings for machine code are described in Usage below

Return value

int. Typical return codes are:

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 *lFlags* 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 as well as the constants to use in a bitwise OR expression to set the option. Bits not listed are reserved.

Bit	Value and meaning	Constant to include in ORed expression
0	0 = Pcode 1 = Machine code	To get machine code, use PBORCA_MACHINE_CODE or PBORCA_MACHINE_CODE_NATIVE
1	0 = Native code 1 = 16-bit code	To get 16-bit machine code, use PBORCA_MACHINE_CODE and PBORCA_MACHINE_CODE_16 To get 16-bit Pcode, use PBORCA_P_CODE_16 **Not supported in PowerBuilder 7** PowerBuilder 7 does not support the Windows 3.x 16-bit platform.
2	0 = No Open Server 1 = Open Server	To build an Open Server executable, use PBORCA_OPEN_SERVER **Not supported after PowerBuilder 5** The OpenClientServer driver was no longer supported after PowerBuilder 5. Therefore, the Open Server executable option is no longer supported.
4	0 = No trace information 1 = Trace information	To get trace information, use PBORCA_TRACE_INFO
5	0 = No error context 1 = Error context	To get error context information, use PBORCA_ERROR_CONTEXT Error context provides the script name and line number of an error
8	0 = No optimization 1 = Optimization	See Bit 9

Bit	Value and meaning	Constant to include in ORed expression
9	0 = Optimize for speed 1 = Optimize for space	To optimize the executable for speed, use PBORCA_MACHINE_CODE_OPT or PBORCA_MACHINE_CODE_OPT_SPEED To optimize the executable for space, use PBORCA_MACHINE_CODE_OPT and PBORCA_MACHINE_CODE_OPT_SPACE

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

```
lFlags = PBORCA_P_CODE;
```

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

```
lFlags = PBORCA_MACHINE_CODE |
      PBORCA_MACHINE_CODE_OPT |
      PBORCA_MACHINE_CODE_OPT_SPACE;
```

Several 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
  ```

- **PBORCA_MACHINE_OPEN_DEFAULT** Meaning native machine code built for Open Server, optimized for speed

  Equivalent to:

  ```
  PBORCA_MACHINE_CODE | PBORCA_OPEN_SERVER |
        PBORCA_MACHINE_CODE_OPT_SPEED
  ```

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:

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

// specify filenames
pszExecFile = "c:\\app\\process.exe";
pszIconFile = "c:\\app\\process.ico";
pszResourceFile = "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;

//Get the proc address of the callback function
lpLinkErrProc =
   MakeProcInstance((FARPROC)LinkErrors, hInst);

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

FreeProcInstance(lpLinkErrProc);
```

For more information about setting up the data buffer for the callback, see "Content of a callback function" on page 16 and the example for PBORCA_LibraryDirectory.

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also PBORCA_DynamicLibraryCreate

PBORCA_LibraryCommentModify

Description

Modifies the comment for a PowerBuilder library.

Syntax

int **PBORCA_LibraryCommentModify** (HPBORCA *hORCASession*,
LPSTR *lpszLibName*,
LPSTR *lpszLibComments*);

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:

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:

```
LPSTR pszLibraryName;
LPSTR pszLibraryComments;

// Specify library name and comment string
pszLibraryName =
   "c:\\sybase\\pb7\\demo\\master.pbl";
pszLibraryComments =
   "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" on page 26.

See also PBORCA_LibraryCreate

PBORCA_LibraryCreate

Description

Creates a new PowerBuilder library.

Syntax

int **PBORCA_LibraryCreate** (HPBORCA *hORCASession*,
    LPSTR *lpszLibraryName*,
    LPSTR *lpszLibraryComments*);

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the filename 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:

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 don’t 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:

```
LPSTR pszLibraryName;
LPSTR pszLibraryComments;

// Specify library name and comment string
pszLibraryName =
    "c:\\sybase\\pb7\\demo\\newlib.pbl";
pszLibraryComments =
    "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" on page 26.

See also

PBORCA_LibraryDelete

PBORCA_LibraryDelete

Description

Deletes a PowerBuilder library file from disk.

Syntax

int **PBORCA_LibraryDelete** (HPBORCA *hORCASession*, LPSTR *lpszLibraryName*);

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

Return value

int. Typical return codes are:

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

Examples

This example deletes a library called EXTRA.PBL:

```
LPSTR pszLibraryName;

// Specify library name
pszLibraryName = "c:\\sybase\\pb7\\demo\\extra.pbl";

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

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_LibraryCreate

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 hORCASession,
          LPSTR lpszLibName,
          LPSTR lpszLibComments,
          int iCmntsBuffSize,
          PBORCA_LISTPROC pListProc,
          LPVOID pUserData );
```

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibName	Pointer to a string whose value is the filename of the library for which you want directory information
lpszLibComments	Pointer to a buffer in which ORCA will put comments stored with the library
iCmntsBuffSize	Size of the buffer pointed to by lpszLibComments
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:

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 don't 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 don't know how many entries there are in the library. However, you need to provide a buffer for the callback function large enough to hold all the information you want to process. There are various approaches you can take:

- Allocate a large amount of memory and hope it is enough, dropping messages that don't fit (illustrated in "About ORCA callback functions" on page 15).
- Call PBORCA_LibraryDirectory once to count the number of entries and again to collect the information for the entries. Each call requires its own callback function designed for the specific task.
- In the callback function, allocate a larger buffer when the original buffer gets full, resetting pointers appropriately.

Examples

This example sets up an InfoBuffer for holding directory information. The structure ORCA_UserDataInfo stores a pointer to the InfoBuffer in lpszBuffer. It has other fields for keeping track of how many times the callback gets called and how full the InfoBuffer is. The callback function keeps the structure up to date by incrementing dwCallCount and updating the offset after it copies information about a directory entry into the InfoBuffer.

This code declares the structure:

```
typedef struct ORCA_UserDataInfo
{
    LPSTR lpszBuffer; // Buffer for entry info
    DWORD dwCallCount; // # of entries in lib
    DWORD dwBufferSize; // size of buffer
    DWORD dwBufferOffset; // current offset in buffer
} ORCA_USERDATAINFO, FAR *PORCA_USERDATAINFO;
```

This code sets up the UserDataBuffer and InfoBuffer variables and a Proc instance for the callback. Then it calls PBORCA_LibraryDirectory. The function stores the library comments in szComments:

```
ORCA_USERDATAINFO UserDataBuffer;
PORCA_USERDATAINFO lpUserDataBuffer;
CHAR InfoBuffer[60000];
LPSTR pszLibraryName;
CHAR szComments[PBORCA_MAXCOMMENT];

// Set library name
pszLibraryName = "c:\\myapp\\ver1\\datamod.pbl";

// Initialize UserDataBuffer
lpUserDataBuffer = &UserDataBuffer;

lpUserDataBuffer->dwCallCount = 0;
lpUserDataBuffer->dwBufferOffset = 0;
lpUserDataBuffer->dwBufferSize = 60000;
lpUserDataBuffer->lpszBuffer = (LPSTR) InfoBuffer;

// Initialize all of InfoBuffer to 0
memset(lpUserDataBuffer->lpszBuffer,
   0x00, (size_t) lpUserDataBuffer->dwBufferSize);

// Get the proc address of callback function
lpListProc = MakeProcInstance(
   (FARPROC)LibraryList, hInst);

// Call ORCA function
lpORCA_Info->lReturnCode = PBORCA_LibraryDirectory(
   lpORCA_Info->hORCASession, pszLibraryName,
   (LPSTR) szComments, PBORCA_MAXCOMMENT,
   (PBORCA_LISTPROC) lpListProc, lpUserDataBuffer);

//Release the proc instance resource
FreeProcInstance(lpListProc);
```

For more information about setting up the data buffer for the callback, see "Content of a callback function" on page 16.

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_LibraryEntryInformation

PBORCA_LibraryEntryCopy

Description

Copies a PowerBuilder library entry from one library to another.

Syntax

int **PBORCA_LibraryEntryCopy** (HPBORCA *hORCASession*,
LPSTR *lpszSourceLibName*,
LPSTR *lpszDestLibName*,
LPSTR *lpszEntryName*,
PBORCA_TYPE *otEntryType*);

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszSourceLibName	Pointer to a string whose value is the filename of the source library containing the object
lpszDestLibName	Pointer to a string whose value is the filename 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:

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

Examples

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

```
lrtn = PBORCA_LibraryEntryCopy(
    lpORCA_Info->hORCASession,
    "c:\\app\\source.pbl", "c:\\app\\destin.pbl",
    "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_LibraryEntryCopy(
    lpORCA_Info->hORCASession,
    lpszSourceLibraryName, lpszDestinationLibraryName,
    lpszEntryName, otEntryType );
```

See also

PBORCA_LibraryDelete
PBORCA_LibraryEntryMove

PBORCA_LibraryEntryDelete

Description

Deletes a PowerBuilder library entry.

Syntax

int **PBORCA_LibraryEntryDelete** (HPBORCA *hORCASession*,
LPSTR *lpszLibName*,
LPSTR *lpszEntryName*,
PBORCA_TYPE *otEntryType*);

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibName	Pointer to a string whose value is the filename of the library containing the object
lpszEntryName	Pointer to a string whose value is the name of the object being deleted
otEntryType	A value of the PBORCA_TYPE enumerated data type specifying the object type of the entry being deleted. 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:

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 don't 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:

```
lrtn = PBORCA_LibraryEntryDelete(
      lpORCA_Info->hORCASession,
      "c:\\app\\source.pbl",
      "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->hORCASession,
      lpszLibraryName, lpszEntryName, otEntryType);
```

See also

PBORCA_LibraryEntryCopy
PBORCA_LibraryEntryMove

PBORCA_LibraryEntryExport

Description

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

Syntax

```
int PBORCA_LibraryEntryExport ( HPBORCA hORCASession,
        LPSTR lpszLibraryName,
        LPSTR lpszEntryName,
        PBORCA_TYPE otEntryType,
        LPSTR lpszExportBuffer,
        LONG lExportBufferSize );
```

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the filename 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_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT PBORCA_BINARY
lpszExportBuffer	Pointer to a buffer in which ORCA will store the source code for the exported object
lExportBufferSize	Size of the buffer pointed to by lpszExportBuffer

Return value

int. Typical return codes are:

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

Usage

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

How the source code is returned The object’s source code is returned in the export buffer. The comparable function in the Library painter saves the exported source in a text file.

The Library painter includes two header lines in the file. ORCA does not add header lines in its export buffer.

In the buffer, the exported source code includes carriage return (hex 0D) and newline (hex 0A) characters at the end of each display line.

Size of source code To find out the size of the source for an object, call the PBORCA_LibraryEntryInformation function first and use the source size information to set the size of the export buffer. If you don’t provide a big enough buffer, the function returns an error code.

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. On export, the buffer is filled with the hexascii representation of binary data.

To check whether there is binary data to export, use the PBORCA_LibraryEntryInformation function with the PBORCA_BINARY type. If the function doesn’t return an error, then you know there is binary data and you know the size of the buffer required to hold it.

Examples

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

```
char szEntrySource[60000];

lrtn = PBORCA_LibraryEntryExport(
    lpORCA_Info->hORCASession,
    "c:\\app\\source.pbl",
    "d_labels", PBORCA_DATAWINDOW,
    (LPSTR) szEntrySource, 60000);
```

This example demonstrates exporting first a window and then the embedded Word file contained in the window. You must use the PBORCA_BINARY type when exporting embedded binary information such as OLE objects:

```
if ( PBORCA_LibraryEntryExport(hOrca, szPblPath,
                   szLibEntry,
                   PBORCA_WINDOW,
                   szExportBufferOne,
                   MAX_EXPORT_SIZE) )
```

```
   {
      PBORCA_SessionGetError(hOrca, szBuffer, 256);
      szMsg = "ORCA Error Message: ";
      szMsg += szBuffer;
      szMsg += "\r\nExport Failed.";
      SetOutputText(szMsg, 2);
      PBORCA_SessionClose(hOrca);
      return;
   }
else
      SetOutputText("Export successful.\r\n", 1);

CString szFile = m_szWorkingDir + "\\" +
m_szLibEntryNameOne +".tmp";

CFile cExpFile(szFile, CFile::modeCreate |
CFile::modeWrite);

cExpFile.Write(szExportBufferOne,
strlen(szExportBufferOne));

cExpFile.Close();
szMsg = "The exported file has been written to: ";
szMsg += szFile + "\r\n";
szMsg += "If you open the file, you will notice that
binary info of the OLE object was not exported by
ORCA.\r\n";
SetOutputText(szMsg, 3);

if ( PBORCA_LibraryEntryExport(hOrca, szPblPath,
                 szLibEntry,
                 PBORCA_BINARY,
                 szExportBufferOneBin,
                 MAX_EXPORT_SIZE) )
   {
      PBORCA_SessionGetError(hOrca, szBuffer, 256);
      szMsg = "ORCA Error Message: ";
      szMsg += szBuffer;
      szMsg += "\r\nExport Failed.";
      SetOutputText(szMsg, 2);
      PBORCA_SessionClose(hOrca);
      return;
   }
   else
      SetOutputText("Export successful.\r\n", 1);
```

See also PBORCA_CompileEntryImport

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*,
LPSTR *lpszLibraryName*,
LPSTR *lpszEntryName*,
PBORCA_TYPE *otEntryType*,
PPBORCA_ENTRYINFO *pEntryInformationBlock*);

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the filename 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 PBORCA_TYPE enumerated data type specifying the object type of the entry. Values are: PBORCA_APPLICATION PBORCA_DATAWINDOW PBORCA_FUNCTION PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE 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:

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 don’t 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 ORCA_EntryInfo
{
   DWORD dwObjectSize; // size of object
   DWORD dwSourceSize; // size of source
   LPSTR lpszComments; // ptr to comments buffer
   LPSTR lpszCreateDate; // date last modified
   LPSTR lpszCreateTime; // time last modified
} ORCA_ENTRYINFO, FAR *PORCA_ENTRYINFO;
```

Use for the source code size When you want to get the source code for an object, you need a buffer big enough to hold the source code. You can call PBORCA_LibraryEntryInformation before calling PBORCA_LibraryEntryExport to determine how big a buffer you need.

Using PBORCA_BINARY for entry type Before you export entries that contain embedded OLE objects, use the PBORCA_BINARY type with the PBORCA_LibraryExportInformation function to find out the size of the buffer you need to hold the hexascii representation of the binary data that will be exported.

Examples

This example sets up a PBORCA_ENTRYINFO structure called EntryInformationBlock then calls PBORCA_LibraryEntryInformation to get information about the DataWindow object d_labels. It extracts the size of the object’s source code:

```
PBORCA_ENTRYINFO EntryInformationBlock;
PPBORCA_ENTRYINFO lpEntryInformationBlock;
DWORD dwSourceSize;

// Initialize pointer to entry info structure
lpEntryInformationBlock =
   (PPBORCA_ENTRYINFO) EntryInformationBlock;
// Initialize structure with nulls
memset(lpEntryInformationBlock,
   0x00, sizeof(PBORCA_ENTRYINFO));

// Get information about entry
lpORCA_Info->lReturnCode =
   PBORCA_LibraryEntryInformalpORCA_Info
   ->hORCASession,
   "c:\\app\\source.pbl", "d_labels",
        PBORCA_DATAWINDOW, lpEntryInformationBlock);

    // If there was no error, process the results
    if (lpORCA_Info->lReturnCode == 0)
    {
        dwSourceSize =
        lpEntryInformationBlock->dwSourceSize;
    }
```

See also PBORCA_LibraryDirectory

PBORCA_LibraryEntryMove

Description

Moves a PowerBuilder library entry from one library to another.

Syntax

```
int PBORCA_LibraryEntryMove ( PBORCA hORCASession,
          LPSTR lpszSourceLibName,
          LPSTR lpszDestLibName,
          LPSTR lpszEntryName,
          PBORCA_TYPE otEntryType );
```

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszSourceLibName	Pointer to a string whose value is the filename of the source library containing the object
lpszDestLibName	Pointer to a string whose value is the filename 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 PBORCA_MENU PBORCA_QUERY PBORCA_STRUCTURE PBORCA_USEROBJECT PBORCA_WINDOW PBORCA_PIPELINE PBORCA_PROJECT PBORCA_PROXYOBJECT

Return value

int. Typical return codes are:

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

Examples

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

```
lrtn = PBORCA_LibraryEntryMove(
    lpORCA_Info->hORCASession,
    "c:\\app\\source.pbl", "c:\\app\\destin.pbl",
    "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

PBORCA_ListCheckOutEntries

Description

Returns check-out information for objects in a PowerBuilder library.

Syntax

int **PBORCA_ListCheckOutEntries** (HPBORCA *hORCASession*,
LPSTR *lpszLibraryName*,
PBORCA_CHECKPROC *lpCallbackFunction*,
LPVOID *pUserData*);

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the filename of the library for which you want check-out information
lpCallbackFunction	Pointer to the PBORCA_ListCheckOutEntries callback function. The callback function is called for each entry in the library The information ORCA passes to the callback function is entry name, library name, user ID, and check-out state, stored in a structure of type PBORCA_CHECKOUT
lpUserData	Pointer to user data to be passed to the PBORCA_ListCheckOutEntries 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:

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

Information returned The information you get with PBORCA_ListCheckOutEntries depends on whether you call it for a master library or a work library:

- **Master library** All objects in the library that have been registered or checked out and which work libraries they were checked out to
- **Work library** All objects in the library that have been checked out from a master library and which library they were checked out from

Examples

This example gets check-out information for objects in the library DWOBJECTS.PBL. For each registered entry in the library, PBORCA_ListCheckOutEntries calls the callback ListCheckOut. In the code you write for ListCheckOut, you store the check-out information in the buffer pointed to by lpUserData. In the example, the lpUserData buffer has already been set up:

```
FARPROC lpCheckProc;
int nReturnCode;

// Get the proc address of the callback function
lpCheckProc = MakeProcInstance(
   (FARPROC)ListCheckOut, hInst);

nReturnCode = PBORCA_ListCheckOutEntries(
   lpORCA_Info->hORCASession,
   "c:\\app\\dwobjects.pbl",
   (PBORCA_CHECKPROC) lpCheckProc, lpUserData);

FreeProcInstance(lpCheckProc);
```

For more information about setting up the data buffer for the callback, see "Content of a callback function" on page 16 and the example for PBORCA_LibraryDirectory.

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_CheckInEntry
PBORCA_CheckOutEntry

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,
        LPSTR lpszLibraryName,
        LPSTR lpszEntryName,
        PBORCA_TYPE otEntryType,
        PBORCA_HIERPROC pHierarchyProc,
        LPVOID pUserData );
```

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the filename 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 allowed values 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 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:

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

Return code	Description
-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 doesn't 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:

```
FARPROC lpHierarchyProc;
int nReturnCode;

//Get the proc address of the callback function
lpHierarchyProc = MakeProcInstance(
   (FARPROC)ObjectQueryHierarchy, hInst);

nReturnCode = PBORCA_ObjectQueryHierarchy(
   lpORCA_Info->hORCASession,
   "c:\\app\\windows.pbl",
   "w_processdata", PBORCA_WINDOW,
   (PBORCA_HIERPROC) lpHierarchyProc, lpUserData );

FreeProcInstance(lpHierarchyProc);
```

For more information about setting up the data buffer for the callback, see "Content of a callback function" on page 16 and the example for PBORCA_LibraryDirectory.

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_ObjectQueryReference

PBORCA_ObjectQueryReference

Description

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

Syntax

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

Argument	Description
hORCASession	Handle to previously established ORCA session
lpszLibraryName	Pointer to a string whose value is the filename 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 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 PBORCA_REFERENCE
pUserData	Pointer to user data to be passed to the PBORCA_ObjectQueryReference callback function 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

Return value

int. Typical return codes are:

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
-9 PBORCA_INVALIDNAME	Name doesn't 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:

```
FARPROC lpReferenceProc;
int nReturnCode;

//Get the proc address of the callback function
lpReferenceProc = MakeProcInstance(
   (FARPROC)ObjectQueryReference, hInst);

nReturnCode = PBORCA_ObjectQueryReference(
   lpORCA_Info->hORCASession,
   "c:\\app\\windows.pbl",
   "w_processdata", PBORCA_WINDOW,
   (PBORCA_REFPROC) lpReferenceProc, lpUserData );

FreeProcInstance(lpReferenceProc);
```

For more information about setting up the data buffer for the callback, see "Content of a callback function" on page 16 and the example for PBORCA_LibraryDirectory.

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_ObjectQueryHierarchy

PBORCA_SessionClose

Description

Terminates an ORCA session.

Syntax

void **PBORCA_SessionClose** (HPBORCA *hORCASession*);

Argument	Description
hORCASession	Handle to previously established ORCA session

Return value

None.

Usage

PBORCA_SessionClose will free any currently allocated resources related to the ORCA session. If you don’t close the session, memory allocated by PowerBuilder DLLs will not be 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 session and frees the ORCA DLL:

```
// Close ORCA session, reset stored handle to 0
PBORCA_SessionClose(lpORCA_Info->hORCASession);
lpORCA_Info->hORCASession = 0;

// Free the ORCA library, reset stored handle to 0
FreeLibrary((HWND) lpORCA_Info->hLibrary);
lpORCA_Info->hLibrary = 0;
```

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_SessionOpen

PBORCA_SessionGetError

Description

Gets the current error for an ORCA session.

Syntax

void **PBORCA_SessionGetError** (HPBORCA *hORCASession*, LPSTR *lpszErrorBuffer*, int *iErrorBufferSize*);

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" on page 27. However, you can get ORCA's complete error message by calling PBORCA_SessionGetError.

If there is no current error, the function will put 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" on page 26.

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.

Examples

This example loads the ORCA DLL and opens a session:

```
// Load the ORCA DLL and check the library handle
lpORCA_Info->hLibrary = LoadLibrary("pborc050.dll");

// Open the session
lpORCA_Info->hORCASession = PBORCA_SessionOpen();

// Check for a valid session handle
if (lpORCA_Info->hORCASession == 0)
{
lpORCA_Info->lReturnCode = 8;
ORCAstrcpy(lpORCA_Info->lpszErrorMessage,
"Open session failed");
}
```

See also

PBORCA_SessionClose
PBORCA_SessionSetLibraryList
PBORCA_SessionSetCurrentAppl

PBORCA_SessionSetCurrentAppl

Description

Establishes the current Application object for an ORCA session.

Syntax

int PBORCA_**SessionSetCurrentAppl** (HPBORCA *hORCASession*,
LPSTR *lpszApplLibName*, LPSTR *lpszApplName*);

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:

Return code	Description
0 PBORCA_OK	Operation successful
-1 PBORCA_INVALIDPARMS	Invalid parameter list
-2 PBORCA_DUPOPERATION	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" on page 22.

Examples

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

```
LPSTR pszLibraryName;
LPSTR pszApplName;
intnReturnCode;

// specify library name
pszLibraryName =
   "c:\\sybase\\pb7\\demo\\master.pbl")

// specify application name
pszApplName = "demo";

// set the current Application object
nReturnCode = 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" on page 26.

See also

PBORCA_SessionSetLibraryList

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*, LPSTR far **pLibNames*, int *iNumberOfLibs*);

Argument	Description
hORCASession	Handle to previously established ORCA session
**pLibNames*	Pointer to an array of pointers to strings. The values of the strings are filenames 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:

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

Examples

This example builds an array of library filenames and sets the session's library list:

```
LPSTR lpLibraryNames[2];
int nReturnCode;

// specify the library names
lpLibraryNames[0] =
   "c:\\sybase\\pb7\\demo\\master.pbl";
strcpy(szLibraryNames[1] =
   "c:\\sybase\\pb7\\demo\\work.pbl";

nReturnCode = PBORCA_SessionSetLibraryList(
   lpORCA_Info->hORCASession, lpLibraryNames, 2);
```

In these examples, session information is saved in the data structure ORCA_Info, shown in "About the examples" on page 26.

See also

PBORCA_SessionSetCurrentAppl

CHAPTER 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.

Contents

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 (FAR PASCAL *PBORCA_ERRPROC) (
        PPBORCA_COMPERR, LPVOID );
```

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" on page 15.

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;
        LPSTR lpszMessageNumber;
        LPSTR lpszMessageText;
        UINT iColumnNumber;
        UINT iLineNumber;
} PBORCA_COMPERR, FAR *PPBORCA_COMPERR;
```

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

Usage

A single error may 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:

Lvl	Num	Message text	Col	Line
0	null	Object: f_boolean_to_char	0	0
0	null	Function Source	0	0

Lvl	Num	Message text	Col	Line
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

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 (FAR PASCAL *PBORCA_LISTPROC) (
          PPBORCA_DIRENTRY, LPVOID );
```

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" on page 15.

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 {
        CHAR szComments[PBORCA_MAXCOMMENT + 1];
        LONG lCreateTime;
        LONG lEntrySize;
        LPSTR lpszEntryName;
        PBORCA_TYPE otEntryType;
} PBORCA_DIRENTRY, FAR *PPBORCA_DIRENTRY;
```

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 PBORCA_TYPE specifying the data type of the object

Callback function for PBORCA_ListCheckOutEntries

Description

Called for each entry in the library so that information about the check-out status of the entry can be stored for later display.

Syntax

```
typedef void (FAR PASCAL *PBORCA_CHECKPROC)
        (PPBORCA_CHECKOUT, LPVOID );
```

Argument	Description
PPBORCA_CHECKOUT	Pointer to the PPBORCA_CHECKOUT structure (described next)
LPVOID	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_CHECKOUT 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 the check-out information for all the entries.

For information and examples of coding a callback function, see "About ORCA callback functions" on page 15.

PBORCA_CHECKOUT structure

Description

Reports check-out information about an entry in a library.

The PBORCA_ListCheckOutEntries function passes the PBORCA_CHECKOUT structure to its callback function.

Syntax

```
typedef struct pborca_checkout {
        LPSTR lpszEntryName;
        LPSTR lpszLibraryName;
        LPSTR lpszUserID;
        CHAR cMode;
} PBORCA_CHECKOUT, FAR *PPBORCA_CHECKOUT;
```

Member	**Description**
lpszEntryName	Pointer to a string whose value is the name of the object whose checkout status is being reported
lpszLibraryName	Pointer to a string whose value is the filename of either the master or work library, depending on the value of *cMode*
lpszUserID	Pointer to a string whose value is the user ID for the checked-out object. If the object is not checked out, lpszUserID is 0
cMode	A character indicating the check-out status of the object. Values are: • d — The entry is checked out. The library passed by the PBORCA_ListCheckOutEntries function is the master library, and the library named in the lpszLibraryName member is the work library to which it was checked out • s — The entry is checked out. The library passed by the PBORCA_ListCheckOutEntries function is the work library, and the library named in the lpszLibraryName member is the master library from which it was checked out • r — The entry is registered but not checked out. The library passed by the PBORCA_ListCheckOutEntries function is a master library

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 (FAR PASCAL *PBORCA_HIERPROC) (
          PPBORCA_HIERARCHY, LPVOID );
```

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" on page 15.

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 {
        LPSTR lpszAncestorName;
} PBORCA_HIERARCHY, FAR *PPBORCA_HIERARCHY;
```

Member	**Description**
lpszAncestorName	Pointer to name of ancestor object

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 (FAR PASCAL *PBORCA_REFPROC) (
          PPBORCA_REFERENCE, LPVOID );
```

Argument	Description
PPBORCA_REFERENCE	Pointer to the PBORCA_REFERENCE 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 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" on page 15.

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 {
        LPSTR lpszLibraryName;
        LPSTR lpszEntryName;
        PBORCA_TYPE otEntryType;
} PBORCA_REFERENCE, FAR *PPBORCA_REFERENCE;
```

Member	**Description**
lpszLibraryName	Pointer to a string whose value is the filename of the library containing the referenced object
lpszEntryName	Pointer to a string whose value is the name of the referenced object
otEntryType	A value of the enumerated data type PBORCA_TYPE specifying the type of the referenced object

Callback function for PBORCA_ExecutableCreate

Description

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

Syntax

```
typedef void (FAR PASCAL *PBORCA_LNKPROC) ( PPBORCA_LINKERR,
          LPVOID );
```

Argument	Description
PPBORCA_LINKERR	Pointer to the PBORCA_LINKERR 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 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" on page 15.

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 {
        LPSTR lpszMessageText;
} PBORCA_LINKERR, FAR *PPBORCA_LINKERR;
```

Member	Description
lpszMessageText	Pointer to the text of the error message